

Abstract Machine for \mathcal{LDL}

Danette Chimenti Ruben Gamboa Ravi Krishnamurthy

MCC, 3500 West Balcones Center Drive, Austin, Texas 78759

danette@mcc.com, ruben@mcc.com, ravi@mcc.com

Abstract

We propose an abstract machine for \mathcal{LDL} that maintains a high-level view of an \mathcal{LDL} program while incorporating aspects of its execution that make a performance difference. A canonical AND/OR graph corresponding to the \mathcal{LDL} program provides the skeleton of its execution. The nodes in the AND/OR graph are annotated to specify relevant details of the execution, such as access methods, join methods, execution strategies, intelligent backtracking, etc. We formalize four execution methods (top-down, bottom-up, as well as two hybrid methods that incorporate memoing) and two recursive computations (fixpoint and stack-based). The two computations and four execution methods are combined to cater to a rich variety of recursive techniques. This annotated AND/OR graph represents a declarative program for the abstract machine. The set of all possible annotated AND/OR graphs constitutes the execution space that defines the abstract machine.

To prove the feasibility of this declarative abstract machine, we demonstrate an actual realization by presenting a code generation algorithm that proceeds by translating each node in the annotated AND/OR graph into a sequence of imperative statements that include calls to a tuple-level interface of an underlying DBMS. The \mathcal{LDL} compiler — which supports Datalog, sets, updates, negation, non-deterministic choice and other advanced features — has been implemented using this approach.

Contents

1	Introduction	1
2	Definitions	2
3	Non-Recursive Datalog	3
3.1	Non-recursive Execution Structure	3
3.2	Execution Method Annotations	3
3.3	Dataflow Annotations	5
3.4	Code Generation	6
3.4.1	Conventions	8
3.4.2	Pipelined Execution	9
3.4.3	Lazy Pipelined Execution	11
3.4.4	Lazy Materialized Execution	12
3.4.5	Materialized Execution	12
3.5	More Annotations: Access methods	12
3.6	More on Dataflow Annotations: Intelligent Backtracking	13
3.7	Execution Space: The Declarative Abstract Machine	15
3.8	Relaxing the Tree Assumption	15
3.9	Implementation Notes	16
4	Recursive Datalog	17
4.1	Execution Method Annotations	17
4.2	Dataflow Annotations	18
4.3	Code Generation	18
4.3.1	Conventions	18
4.3.2	Semi-Naive Computation	20
4.3.3	Stack-Based Computation	22
4.4	Implementation Notes	23
5	Conclusions	24

1 Introduction

The Logic Data Language, \mathcal{LDL} , is a declarative language for data-intensive and knowledge-based applications built upon the experience of both the relational database and logic programming communities [NT89]. The user need only supply the declarative program and the compiler is expected to devise an efficient execution strategy. In order to provide efficient executions for \mathcal{LDL} programs, the compiler must be capable of reasoning about the program at a high level while still retaining knowledge of the underlying system.

In the context of relational programs, join, selection and projection methods together with join ordering provides such a framework. In a sense, the join, selection, and projection methods are instructions in an abstract relational machine. A composition of these instructions that encodes the join ordering constitutes an imperative program to the abstract machine. The program can be viewed as a model of an actual execution, representing the important run-time aspects affecting performance in the underlying system.

Abstract machines have been proposed for logic programming (e.g., the WAM, [Wa83]) by enumerating the instruction set. However, these abstract machines were never intended to provide the same level of abstraction as in the relational approach. Consequently, they do not directly provide a framework to exploit techniques proven successful in database technology, such as effective use of compilation (optimization), use of bottom-up as well as top-down executions, and independence of data through a tuple level interface.

We propose an abstract machine for \mathcal{LDL} that is in the spirit of the relational abstract machine, that is, it incorporates aspects that make a performance difference in execution, while maintaining a high-level of abstraction. We formalize the abstract machine, not by enumerating the instruction set, but by declaring the execution space — the set of all executions that are allowed by the machine. This is achieved by modeling an execution as follows. A canonical AND/OR graph corresponding to an \mathcal{LDL} program provides the skeleton of the execution. Then, the nodes in the AND/OR graph are annotated to specify relevant details of the execution, such as access methods, join methods, execution strategies, etc. This annotated AND/OR graph represents a program for the abstract machine (i.e., an execution model). The set of all possible annotated AND/OR graphs constitutes the execution space that defines the abstract machine.

One of the major contributions of this approach is to present four distinct execution methods — top-down, bottom-up as well as two hybrid methods that incorporate memoing [Mi68] and to demonstrate the usefulness of each. Further, two recursive computations are formalized: the fixpoint (e.g., semi-naive, [Ba85]) and stack-based (e.g., Prolog). These two computations and the four execution methods are combined to cater to a rich variety of recursive techniques ranging from bottom-up using magic sets [BMSU86] to query-subquery using memoing [Vi89] and pure top-down. All these methods are given as declarative annotations to the AND/OR graph. Even though the execution model is proposed in a declarative paradigm, procedural aspects such as intelligent backtracking are also incorporated. To prove the feasibility of this declarative approach, we demonstrate an actual realization by presenting the code generation algorithm used in the \mathcal{LDL} compiler. The algorithm proceeds by translating each node in the annotated AND/OR graph into a sequence of imperative statements that include calls to a tuple-level interface of an underlying DBMS.

The \mathcal{LDL} compiler has been implemented using this abstract machine not only for Datalog, but also to support advanced language features such as sets, updates, negation and non-deterministic choice. In this

paper, however, we restrict our attention to Datalog. We develop the abstract machine for Datalog in two steps: first, for nonrecursive Datalog and then for complete Datalog. The treatise for nonrecursive Datalog is sufficiently encompassing that the recursive case is shown as a natural extension. Therefore, the majority of the paper details the nonrecursive case.

2 Definitions

We assume the reader is familiar with Datalog and the associated semantics, as well as definitions for rule, base/derived predicate, predicate occurrence, and mutual recursion. Throughout this paper, we follow the notational convention that p_i 's and b_i 's are predicates and base predicates, respectively. We define a *clique* to be a maximal set of mutually recursive predicates.¹

A query with marked bound/unbound arguments (binding pattern) will be called a *query form*. Throughout this paper we use capital letters to denote variables and the lower case letter c to denote a constant. Thus, $p1(c,Y)?$ is a query form with the first argument bound and the second unbound. In a departure from previous approaches to compilation of logic, we make our compilation specific to a given query form. For instance, the query form, $p1(X,Y)?$, will be compiled and optimized separately from $p1(c,Y)?$. Indeed the execution strategy chosen for the query $p1(X,Y)?$ may be inefficient for $p1(c,Y)?$, or an execution designed for $p1(c,Y)?$ may not terminate for $p1(X,Y)?$.

In general, given a query form and, for each rule, a specific permutation of the literals in the body, we can define the notion of a binding pattern for any predicate, using sideways information passing (SIP) as defined in [Ull85]. Each argument of the predicate can be either bound, free, or existential. A variable, say X , appearing in a given argument is *bound* if it is instantiated to a particular constant value at run-time. X is *free* if the current predicate occurrence will instantiate it at run-time. Finally, it is *existential* if it does not appear elsewhere in the rule, except possibly in the head as an existential argument. We can construct an *adorned program*, where every argument of each predicate is marked as either b (bound), f (free), or e (existential) [Ull89,RBK88]. Note that for a given query form and permutation of the literals in each rule, there is a unique adornment for the program. As an example, consider the following program:

```
p1(X,Y) <- p2(X,Y), b1(Y,Z).
p2(X,Y) <- b2(X,X1), p2(Y1,X1), b2(Y,Y1).
p2(X,X).
```

The adorned program for the query form $p1(c,Y)?$ is as follows:

```
p1bf(X,Y) <- p2bf(X,Y), b1be(Y,Z).
p2bf(X,Y) <- b2bf(X,X1), p2fb(Y1,X1), b2fb(Y,Y1).
p2bf(X,X).
p2fb(X,Y) <- b2ff(X,X1), p2fb(Y1,X1), b2fb(Y,Y1).
p2fb(X,X).
```

Note that the rules for $p2$ were duplicated for each different adornment of the predicate occurrence. We refer to this process as *stability transformation* [SZ87].

¹Note that a single recursive predicate forms a clique by itself.

3 Non-Recursive Datalog

In this section we develop an abstract machine for non-recursive Datalog (NRD) programs in a declarative fashion. First, we represent the skeleton of the execution by a canonical AND/OR graph, where each OR node represents a predicate occurrence and each AND node represents the head of a rule [Nil80]. Then, we superimpose a set of annotations on the AND/OR nodes which are used to specify the details of the execution, while still retaining the declarative nature of the program. The purpose of the annotated AND/OR graph is to represent those aspects of the execution that can make a performance difference.

Obviously, there are many annotated AND/OR graphs for a given program, all of which can faithfully model the bottom-up semantics of the original NRD program. We define the execution space to be the set of all such faithful executions and use this execution space to define the abstract machine. Any implementation that is capable of computing all the executions in the execution space is a valid implementation of the abstract machine. In order to show that such an implementation exists, we outline the code generation algorithm used in the *LDL* compiler.

3.1 Non-recursive Execution Structure

We can view an NRD program as an AND/OR graph [Nil80]. We first make the assumption that the graph is a tree, possibly by replicating some rules in the program. This assumption is relaxed in section 3.8. As the AND/OR tree and the program have a one-to-one correspondence, we use these terms interchangeably.

In keeping with our relational algebra execution model, we map each AND node into a join and each OR node into a union. Strictly speaking, the relational join operator is binary and a rule containing more than two literals is computed by a sequence of joins. As the order of joins will reflect on the efficiency of the execution, we define a normal form, called *Chomsky Normal Form* or *CNF*² of the program. The CNF program has at most two literals in the body of any rule, which we order left-to-right. It is obvious that such an equivalent program exists. The AND/OR tree corresponding to the CNF program, called the *execution structure*, forms the skeleton of the final execution.³

All variables in the program are assumed to have global scope. That is, when the program is viewed as a logical formula, there is only one (existential or universal) quantifier for each variable. The *LDL* compiler achieves this requirement by renaming as few variables as possible.

3.2 Execution Method Annotations

In this subsection we present four execution methods that capture the notions of bottom-up and top-down, as well as two hybrid executions that incorporate memoing [Mi68]. We also delineate advantages and disadvantages of each method, in order to show that all of these methods are important in the execution model. Each predicate occurrence (i.e., OR node) is annotated with an execution method.

²This form corresponds to the analogous Chomsky Normal Form grammar [HU79].

³A similar transformation can be used for union since it can also be viewed as a binary operator. However, this is not necessary since union ordering is not a significant performance issue.

Let $p(\dots)$ be the query with the following rule, where $p1$ and $p2$ are derived predicates:

$$p(X, Y) \leftarrow p1(X, Z), p2(Z, Y).$$

The four execution methods are described using $p2$ as an example.

- **Pipelined Execution** chooses a tuple for $p1$ which instantiates a value for the variable Z . Only tuples having this Z -value in $p2$ are computed *one tuple at a time* to get the tuples for p .
- **Lazy Pipelined Execution** is a pipelined execution in which, as the tuples are generated for $p2$, they are stored in a temporary relation, say $rp2$, for subsequent use. For each tuple in $p1$, $rp2$ is checked to see if the corresponding tuples in $p2$ have already been computed. If so, the tuples are read from $rp2$ as if it were a base relation; otherwise, the tuples are computed as in the pipelined execution, i.e., one tuple at a time. As the set of tuples computed for a given value for the bound arguments, i.e., Z -value, can be incomplete due to intelligent backtracking (see section 3.6), a *done* indicator is associated with each Z -value to denote the completion of the computation. This also allows for the efficient reuse of *negative information* in the case that no tuple in $p2$ has this Z -value.
- **Lazy Materialized Execution** proceeds as in the lazy pipelined case except that, for a given Z -value, all tuples in $p2$ having that Z -value are computed and stored in $rp2$ before proceeding. The *done* indicator is also used in this case to allow the reuse of negative information.
- **Materialized Execution** computes *all* tuples in $p2$ and stores them in the relation $rp2$. Then, the computation proceeds, using the tuples from $rp2$.⁴

Note that the above discussion can be generalized to any OR node with a (possibly empty) set of bound arguments. We refer to the above executions using the acronyms P, LP, LM and M, respectively. Their relative advantages (+) and disadvantages (−) are summarized in the following table:

	P	LP	LM	M
Amortized Work	−	+	+	+
Superfluous Work	+	+	+	−
Backtrackable	+	+	−	−
Reentrant	−	−	+	+
Space Required	++	−	−	−−

Amortized Work: The materialized (as well as LM and LP) execution computes a tuple, say ta , in $p2$ exactly once and uses it as many times as needed to join with tuples from $p1$, whereas the pipelined execution must recompute the tuple each time. This savings is termed *amortized work*.

Superfluous Work: A tuple, tb , may be computed for $p2$ that is never used to join with any tuple in $p1$. This is referred to as *superfluous work*. The pipelined (as well as LP and LM) execution avoids this by using the SIP property, which specifies that $p2$ tuples are computed for a given value of Z . On the other hand, the materialized execution computes all $p2$ tuples, and, hence, may perform superfluous work.

Backtrackable: The ability to compute the tuples of $p2$ one tuple at a time (as in P and LP) has the advantage that if the binding from $p1$ does not join with other predicates to produce an answer, the compu-

⁴If $p2$ is a base relation, nothing is done to materialize it.

tation for other tuples of $p2$ with the same Z -value can be avoided. This advantage is called *backtrackable advantage*. The lazy materialized (and M) execution computes all the tuples for a Z -value and is not capable of avoiding such useless computations.

Reentrant: If a predicate is used in more than one rule, the code generated for it may be shared in both occurrences. Conceptually, this can be done if the code generated is *reentrant*. In the case of pipelined (and LP) execution, the state of the computation is needed to get the next tuple on backtracking. Therefore, in order to make the code reentrant, the state needs to be managed, resulting in some run-time overhead. On the other hand, the materialized (and LM) execution completes the computation of a predicate before proceeding, allowing multiple occurrences to share the materialized relation.

Space Required: The pipelined execution has the least space requirement, whereas the materialized execution has the most. The lazy executions can be better than the materialized one if there are lots of superfluous tuples, but are not better in the worst case.

In conclusion, the pipelined execution is useful if the joining column is a key for $p1$, whereas the materialized execution is the best if all the Z -values of $p2$ are joined with some $p1$ tuple. Note that in both of these cases, the respective lazy evaluation incurs more overhead due to the checking that is needed for each $p1$ tuple as well as the *done* indicator that must be maintained. The reentrant property is especially useful if the predicate is in the scope of a recursive query that is being computed top-down. Therefore, in such cases LM is preferred over LP. LP is preferable, otherwise, to exploit the backtrackable property.

Interestingly, many researchers assume the computation of a logic program with bottom-up semantics (e.g., \mathcal{LDL}) is necessarily computed bottom-up.⁵ In \mathcal{LDL} , the resulting execution need not be bottom-up. It is usually some combination of top-down and bottom-up. In fact, for NRD programs, the execution is almost always top-down.

3.3 Dataflow Annotations

In this section, we introduce the concept of *dataflow points* to a CNF program. Each node of the corresponding AND/OR tree has dataflow points, which represent different states of computation in the node. The dataflow points are analogous to the ports in Byrd's Prolog execution model [By80]. These points are connected by *dataflow destinations* (referred to in the literature as continuations [Wa71]). The dataflow points and destinations together describe how tuples are combined to produce an answer to a query.

The dataflow points associated with each node of the AND/OR tree and their corresponding state are given in the following table:

DATAFLOW POINT	STATE OF COMPUTATION
entry	getting first tuple of node
backtrack	getting next tuple of node
success	a tuple has been generated
fail	no more tuples can be generated

A dataflow point of one node can be directed to a dataflow point of a different node by a dataflow destination.

⁵This was the single most asked question as well as the source of confusion among the attendees of SIGMOD 1989, in Portland, OR, who came to see the \mathcal{LDL} demonstration.

The *entry destination* (e_dest) of a given node is the dataflow point to which its entry point is directed. Similarly, *backtrack* (b_dest), *success* (s_dest), and *fail destinations* (f_dest) can be defined. The dataflow destinations represent logical operations between the nodes involved; for example, a join of the two nodes.

The dataflow points and destinations of a node describe how the tuples of that node are combined with tuples from other nodes, not how these tuples are generated in the first place. That is, they describe the dataflow of the program, without assuming an execution method (i.e., P, LP, LM or M). In the following description, we use the intuition of a pipelined execution. However, we emphasize that neither the dataflow points nor the destinations are changed for a different execution method.

Figures 1 and 2 show the dataflow points and destinations of a typical OR node and AND node, respectively. In a CNF program, each rule has at most two OR nodes, which are ordered left-to-right, so as to define first and second node. We arbitrarily order the AND node successors of a given OR node, so that the notions of first, next, and last AND node are also defined.

Intuitively, to get the first tuple from an OR node, we get the first tuple from its first successor AND node. To get the next tuple, we get the next tuple from the AND node that generated the previous tuple. Note that the currently “active” AND node must be determined at run-time. Clearly, when a tuple is generated by any AND node, we get a tuple for the OR node. When no more tuples can be generated for a given AND node, we try to generate tuples from the next AND node, until the last such node is reached. At this point, no more tuples can be generated for the OR node.

The execution of an AND node is conceptually less complicated. Intuitively, the execution corresponds to a nested loop, where, for each tuple of the first OR node, we generate all matching tuples from the second OR node. Thus, when generating the next tuple of an AND node, we generate the next matching tuple from the second OR node. If there are no more matching tuples, we generate the next tuple from the first OR node. When there are no more tuples for this OR node, we can generate no more tuples for the AND node.

The execution described above is nicely captured by the following list of destination annotations:⁶

OR node:	AND node:
e_dest : entry point of first AND successor	e_dest : entry point of first OR successor
b_dest : backtrack point of “active” AND successor	b_dest : backtrack point of last OR successor
f_dest : if node is first OR node in a rule then fail point of parent AND node	f_dest : if node is last AND successor then fail point of parent OR node
else backtrack point of previous OR node	else entry point of next AND node
s_dest : if node is last OR node in a rule then success point of parent AND node	s_dest : success point of parent OR node
else entry point of next OR node	

3.4 Code Generation

So far, we have provided a model for the execution of an NRD program by describing an adorned CNF program that is annotated with execution methods and dataflow destinations. The annotated CNF program

⁶Note that for a base relation, the entry and backtrack destinations are not used and, thus, need not be defined.

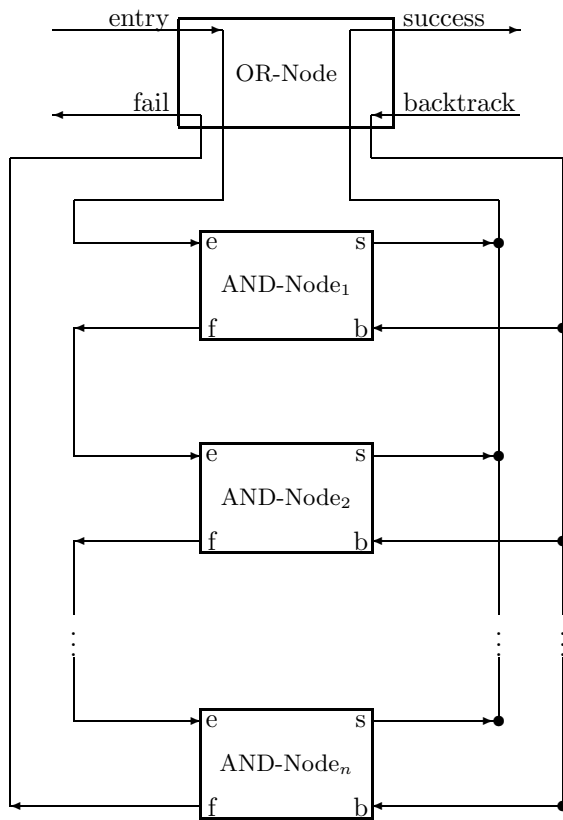


Figure 1: Dataflow Annotations for OR Node

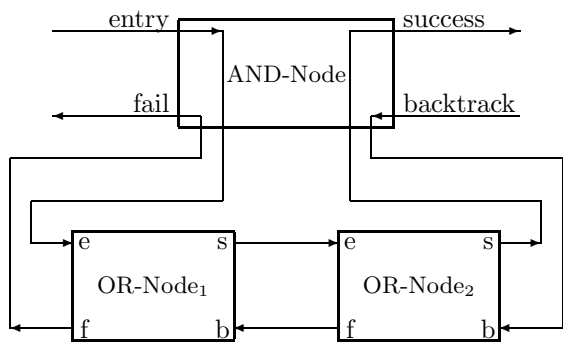


Figure 2: Dataflow Annotations for AND Nodes

is a declarative representation of the final execution. In this section, we turn to an operational view in order to provide the rationale for the particular model chosen, as well as to provide a conceptual view of an actual execution. We show how a CNF program can be compiled into an imperative language by demonstrating that procedural code can be generated which, in fact, reflects all the declarative notions that have been described and is faithful to the dataflow paradigm. We also demonstrate that the annotated CNF program encodes all the information necessary to generate code locally with respect to each node in our execution structure and, thus, allows for a simple, elegant, recursive algorithm to generate efficient run-time code.

We will first outline the conventions necessary to map an annotated CNF program into procedural code. Then we describe the code generation for a pipelined execution, followed by the changes necessary to realize lazy pipelined, lazy materialized and materialized executions.

3.4.1 Conventions

The specific target language of the compiler is largely irrelevant; we assume only that it have `goto` and `indirect goto` statements, as well as other procedural and data representation capabilities available in most imperative programming languages.⁷

A dataflow point of a node corresponds to a label in the compiled program. This label can be chosen with a simple naming convention; for example, `node_e` and `node_s` are the entry and success points of `node`. The `node` portion of the label can be constructed by combining the predicate name and some unique identifier.⁸ Thus, the entry point of an AND node for predicate `p` may be called `p_and_31_e`, where `and_31` is the unique identifier for the AND node. Dataflow destinations can be implemented as simple `goto` statements that jump to the specified dataflow point. However, recall that in the case of an OR node, it is necessary to alter the backtrack destination at run-time. This can be achieved by using an `indirect goto` statement. We need only choose a variable that will store the indirect label. Since only one such variable is needed per OR node, we can call it `node_loc`. Associated with each node is a `node table` entry which contains information relevant to that node. The `node_loc` to be used by a particular node may be stored in the node table entry for that node. Other information stored in the node table will be discussed in the context of recursive Datalog.

Variables in the CNF program are mapped into variables in the compiled program using a simple approach — for example, `X` can be compiled into `var_X`. Recall that all variables in a CNF program have global scope. Thus, this simple naming scheme avoids name conflicts. Moreover, it conveys the global dataflow of the program in a localized manner, since the node assigning `var_X` does not need to be aware of the node that uses this variable in order to store a value. One advantage of our approach is that all variables to be used in the program are known at compile-time, and, hence, space for them can be allocated statically.

Finally, we assume a tuple-level interface to the database. This is achieved through a set of library routines, such as `open_cursor`, `get_tuple`, `create_index`, etc.

⁷Our implementation compiles into *C*, substituting a `switch` statement for the non-existent *C* `indirect goto`.

⁸The unique identifier is necessary because there may be multiple occurrences of the predicate.

3.4.2 Pipelined Execution

We can completely describe the code generation for the pipelined execution of an NRD program by outlining the code that is generated at each of the four dataflow points for a base relation and a derived relation. For a base relation, we need only outline the code for an OR node, while in the case of the derived relation, we outline the code for both an OR node and an AND node.

The code generated at the entry point for a base relation must first initialize a cursor (pointer) on the base relation and then get a tuple from the relation. The code must continue getting tuples until one is found that matches all arguments adorned as bound for this OR node. After obtaining a valid tuple, any arguments which are adorned as free are assigned values from the tuple and a jump to the success point for this node is executed.⁹ If no valid tuples are found, then a jump to the fail point for this node is executed. The code to actually get a tuple from the relation is also needed at the backtrack point. Hence, the entry point simply opens the cursor and then falls into the backtrack point. The code generated for the fail and success points are simply jumps to the node's fail and success destinations, respectively.

- **OR node — base relation:**

```
node_e: open a cursor on the base relation
node_b: get the next tuple that satisfies all bound arguments
        if a tuple is found then
            assign values to free arguments
            goto node_s
        else goto node_f
node_f: goto f_dest for node
node_s: goto s_dest for node
```

At each dataflow point for both a derived relation OR node and an AND node, the code generated is simply a jump to the label for that dataflow destination.¹⁰ The only exception occurs at the backtrack point of the OR node. Here the jump is an indirect jump through the node's backtrack location variable, *node_loc*, as described in section 3.3. This variable is set at run-time at the entry points for each AND node successor to the OR node so that the rule which generated the last tuple will become the backtrack destination for the OR node. Note that each AND node successor must have access to the name of the parent's backtrack location variable. The node table can be used to obtain this information.

- **OR node — derived relation:**

```
node_e: goto e_dest for node
node_b: goto node_loc; i.e., indirect
node_f: goto f_dest for node
node_s: goto s_dest for node
```

- **AND node:**

```
node_e: set parent_node_loc to be node_b
        goto e_dest for node
node_b: goto b_dest for node
node_f: goto f_dest for node
node_s: goto s_dest for node
```

Figure 3 shows an annotated CNF program, the node table and the corresponding code that is generated for the program. We assume an adornment of **bf** for **p1**. In the annotated CNF program, **AD** is the adornment and **LABEL** is the unique identifier. The remaining annotations are the dataflow destinations. Note that the execution method has been omitted from the annotations since we are pipelining all OR nodes.

⁹Note that arguments marked as existential in the adornment are not assigned from the tuple and, in fact, do not affect the code generated at all.

¹⁰Clearly peephole optimization can remove vacuous goto statements — see section 3.9.

Annotated CNF

	AD	LABEL	E_DEST	B_DEST	F_DEST	S_DEST
p1(X,Z) <- p2(X,Y), b1(Y,Z).	bf	p1_and_1	p2_or_2_e	b1_or_7_b	parent_f	parent_s
	bf	p2_or_2	p2_and_3_e	p2_or_2_loc	p1_and_1_f	b1_or_7_e
	bf	b1_or_7	—	—	p2_or_2_b	p1_and_1_s
p2(X,Y) <- b2(X,Y).	bf	p2_and_3	b2_or_4_e	b2_or_4_b	p2_and_5_e	p2_or_2_s
	bf	b2_or_4	—	—	p2_and_3_f	p2_and_3_s
p2(X,Y) <- b3(X,Y).	bf	p2_and_5	b3_or_6_e	b3_or_6_b	p2_or_2_f	p2_or_2_s
	bf	b3_or_6	—	—	p2_and_5_f	p2_and_5_s

Node Table

NODE	LOC
p2_and_3	p2_or_2_loc
p2_and_5	p2_or_2_loc

Generated Code

```

p1_and_1_e: goto p2_or_2_e
p1_and_1_b: goto b1_or_7_b
p1_and_1_f: goto parent_f
p1_and_1_s: goto parent_s

p2_or_2_e: goto p2_and_3_e
p2_or_2_b: indirect goto p2_or_2_loc
p2_or_2_f: goto p1_and_1_f
p2_or_2_s: goto b1_or_7_e

p2_and_3_e: set p2_or_2_loc = p2_and_3_b
            goto b2_or_4_e
p2_and_3_b: goto b2_or_4_b
p2_and_3_f: goto p2_and_5_e
p2_and_5_s: goto p2_or_2_s

b2_or_4_e: open a cursor on b2
b2_or_4_b: get next tuple with 1st arg = X
            if a tuple is found then
                assign Y
                goto b2_or_4_s
            else goto b2_or_4_f
b2_or_4_f: goto p2_and_3_f
b2_or_4_s: goto p2_and_3_s

p2_and_5_e: set p2_or_2_loc = p2_and_5_b
            goto b3_or_6_e
p2_and_5_b: goto b3_or_6_b
p2_and_5_f: goto p2_or_2_f
p2_and_5_s: goto p2_or_2_s

b3_or_6_e: open a cursor on b3
b3_or_6_b: get next tuple with 1st arg = X
            if a tuple is found then
                assign Y
                goto b3_or_6_s
            else goto b3_or_6_f
b3_or_6_f: goto p2_and_5_f
b3_or_6_s: goto p2_and_5_s

b1_or_7_e: open a cursor on b1
b1_or_7_b: get next tuple with 1st arg = Y
            if a tuple is found then
                assign Z
                goto b1_or_7_s
            else goto b1_or_7_f
b1_or_7_f: goto p2_or_2_b
b1_or_7_s: goto p1_and_1_s

```

Figure 3: Annotated CNF, Node Table and Generated Code

3.4.3 Lazy Pipelined Execution

The lazy pipelined execution is useful only for derived relations and, thus, only changes to the code generated for a derived OR node and AND node need to be discussed. Furthermore, we can localize all necessary changes to the OR node and keep the same code for the AND node as shown in the previous section.

In order to generate code for a lazy pipelined execution, a temporary relation (referred to below as the materialized relation) must be used to store and access tuples. In addition, we need to introduce a secondary relation to store a *done* indicator for each binding pattern that has been completely materialized, i.e., all possible tuples with this binding pattern have been computed and stored.

The code at the entry point for the OR node checks to see if the *done* indicator exists for the given bound arguments. If so, a state variable, *node_state*, is set to indicate that tuples are being read from the materialized relation, and a cursor into this relation is opened. Then, the code proceeds to the backtrack point and reads the first tuple from the relation. If, on the other hand, the *done* indicator does not exist, *node_state* is set to indicate that tuples are being materialized and there is a jump to the entry destination.

The code at the backtrack point checks *node_state* and, if the materialized relation is currently being read, it reads the next tuple from the relation just as at the backtrack point for a base relation. If, instead, the tuples are being materialized, there is an indirect jump to *node_loc* as described in the previous section.

In the case where tuples are being materialized, they are stored at the success point of the OR node. Likewise, the *done* indicator is stored at the fail point of the OR node. Note that the fail and success points will only be reached during materialization since the fail and success destinations have been incorporated into the entry and backtrack points where appropriate. Note also that even if no tuples are found for a particular binding pattern, the *done* indicator will be stored at the fail point and subsequent attempts to get tuples with this binding pattern (at the entry point of the OR node) will result in immediate failure. Thus, the *done* indicator also provides *negative information*, avoiding unnecessary computation.

- **OR node:**

```
node_e: if done materializing for the given bound arguments then
        set node_state to reading
        open a cursor on the materialized relation
    else
        set node_state to materializing
        goto e_dest for node
node_b: if node_state is reading then
        get the next tuple that satisfies all bound arguments
        if a tuple is found then
            assign values to free arguments
            goto s_dest for node
        else goto f_dest for node
    else goto node_loc; i.e., indirectly through this variable
node_f: store done for current binding in secondary relation
        goto f_dest for node
node_s: store a tuple (assembled from global variables) in the materialized relation
        goto s_dest for node
```

3.4.4 Lazy Materialized Execution

As with the lazy pipelined case, lazy materialization makes sense only for derived relations and we can localize all changes to the derived OR node. In fact, for the lazy materialized execution, the entry and backtrack points are identical to those given for the lazy pipelined execution. The success and fail points, however, must be altered so that all the tuples corresponding to the given binding pattern are generated before going to the success destination. This is accomplished by jumping to the backtrack point after storing the computed tuple in the materialized relation at the success point. Then, after storing the *done* indicator at the fail point, a jump is made back to the entry point so that the stored tuples may be read.

- **OR node:**

- node_f*: store *done* for current binding in secondary relation

- goto *node_e*

- node_s*: store a tuple (assembled from global variables) in the materialized relation

- goto *node_b*

3.4.5 Materialized Execution

The materialized execution may be viewed simply as a special case of the lazy materialized execution where no arguments are bound. Therefore, no changes need be made to the code outlined for lazy materialization except to note that storing a *done* indicator is superfluous for the materialized execution.

3.5 More Annotations: Access methods

In the last section we digressed to impart the operational view of the declarative execution model consisting of the CNF program attributed with the annotations. Let us return to the declarative model of execution and discuss other annotations that are needed to fully describe the execution.

The reader may have already observed that the CNF program encodes the ordering of joins but not the choice of access methods. We can associate other annotations that detail the choice of access methods with the nodes in the AND/OR tree [Ull88]. Some of these choices are given below to exemplify the aspects of the computation that are modeled as annotations. We also describe the necessary changes to the code generator.

- **Preselect:** The preselect conditions are those bound conditions used to preselect a subset of the dataflow through a node. If the node is an OR node, the conditions select a subset of the tuples in the predicate. The selected tuples are materialized as a temporary relation in a preprocessing phase. Hence, the predicate must be either a base relation or a materialized derived predicate.

If the node is an AND node then these conditions are selections implied by the unification with its parent OR node. These conditions are applied on the fly to the dataflow through this node by incorporating the checks at the entry point.

- **Postselect:** The postselect conditions are all those conditions that can be checked at this node that are not already implied by the preselect, index, etc. If the node is an OR node then checks are incorporated at the entry and backtrack points after getting a tuple.

The set of conditions that are implied by the unification of an AND node with its parent OR node are assignments to free arguments, and are incorporated at the success point of the AND node.

- **Join Methods:** The join method to be used in joining $p1$ and $p2$ is annotated to the OR node corresponding to $p2$, i.e., in the inner-most loop. Therefore, the annotation for any join method is provided by specifying the access method to the $p2$ predicate.¹¹ Needless to say, an access method to $p2$ makes sense only in the case of base relations or derived predicates that are not being pipelined.
- **Define Index:** An access method that uses an index that has to be created (e.g., in the materialized case) is specified using a define index annotation, where the argument(s) and type of index (if many are available) are specified. Obviously, this is applicable only in the case of base relations or derived predicates that are not pipelined. Code is generated to maintain the index as tuples are inserted.

In short, we have provided a sample of annotations that can be attributed to an execution structure (i.e., CNF program) to flesh out the details of the actual execution. The choice of modeling an aspect of the execution as an annotation, as opposed to in the execution structure, has not been arbitrary. Obviously, preselect and postselect can be modeled in the execution structure by rewriting the program. This was not done because of the following observation:

Given an execution structure, the specific annotation can be chosen using a greedy algorithm. This implies that the optimal choice of the annotations does not require an exponential search, whereas the optimal choice of the execution structure does. Therefore, the desiderata that is proposed for the choice of modeling an aspect of execution is to guarantee the existence of an efficient algorithm for the optimal choice.

3.6 More on Dataflow Annotations: Intelligent Backtracking

One important aspect of the execution that needs to be represented in the execution model is the capability of intelligent backtracking. This is easily done in the context of the dataflow paradigm by changing the dataflow destinations.

Intelligent backtracking as proposed in the context of Prolog has been known to require too much run-time overhead [PP82]. As a result, in most cases, it does not reduce the execution time. This is because intelligent backtracking is not applicable for most joins. Consequently, even a small overhead to detect that intelligent backtracking is unnecessary will add up to total more than the savings. For this reason, in *LDL* we incorporate backtracking techniques by doing compile-time analysis such that little (if any) overhead is incurred at run-time.

Intelligent backtracking consists of avoiding useless computation that cannot generate new tuples. Initially, we assume only a single rule per predicate. We can classify the useless computations as follows. Consider the following rule, with all three OR nodes to be pipelined:

$$p(X, Y) \leftarrow p1(X, Z), p2(Y, Z), p3(X, U).$$

¹¹In the case of sort merge, an appropriate annotation for sorting $p1$ is required for the execution to be valid.

- **Get-First Optimization:** If the attempt to get the first tuple in $p3$ fails, it is unnecessary to backtrack to $p2$, since it does not change the bound argument for $p3$, i.e., X . The only way this value can be changed is by backtracking to $p1$. In general, on entry to an OR node, if no tuples are found, we need to backtrack to the OR node where the bound variable is instantiated (i.e., is adorned as free).
- **Get-Next Optimization:** After computing a tuple for p , backtracking to get the next tuple for $p3$ is unnecessary, since it will not yield any new tuples for p . A new p tuple can only be generated by backtracking to $p2$. In general, when backtracking from an AND node, the computation should resume at the last OR node that binds a variable occurring in the head.

These two optimizations can be incorporated into the dataflow annotations by providing two fail destinations for each node: one for the entry and the other for the backtrack point of the node. We define these dataflow destinations as follows:

- *Entry Fail Destination:* For an OR node with exactly one bound argument,¹² the entry fail destination is the backtrack point of the first OR node where the argument is adorned as free, if it occurs in the same rule, and the entry fail destination of the head, otherwise. For an AND node, it is the entry destination of the parent OR node.
- *Backtrack Fail Destination:* For an OR node, the backtrack fail destination is the backtrack point of the previous OR node in the rule that binds a variable occurring in the head, if such an OR node exists, and the backtrack fail destination of the parent AND node, otherwise. For an AND node, it is the backtrack point of the last OR node in the rule that binds a variable occurring in the head if such an OR node exists, and the backtrack fail destination of the parent OR node, otherwise.

Note that the backtrack fail destination corresponds to the previous notion of fail destination, which is now subsumed. Also note that, the entry fail destination can only be used if we detect failure of the node at the entry point as well as the backtrack point for the node.

It is easy to show that the above annotations result in an execution that faithfully models the bottom-up semantics of the program. Note that there is absolutely no run-time overhead involved in the above changes and that these changes will result in improved executions.

Allowing more than one rule per predicate poses the problem of detecting the get-first failure for all rules of the derived predicate. This can be achieved by keeping a count of all AND nodes that register an entry fail, so that the last AND node can determine if get-first optimization is applicable. This incurs a marginal overhead (a single assignment per rule).¹³ This overhead is only incurred for derived predicates with multiple rules. No additional overhead is incurred for get-next optimization. We note that compile-time analysis can be used to avoid intelligent backtracking in cases where it is of little (or no) advantage. For example, when the entry fail destination of a node is the same as its backtrack fail destination.

¹²We make this assumption for ease of exposition. The \mathcal{LDL} compiler allows arbitrary binding patterns in the OR node.

¹³The overhead was observed to be insignificant in the \mathcal{LDL} implementation.

3.7 Execution Space: The Declarative Abstract Machine

Using the annotations described in the previous subsections we can now define an *annotated CNF program* to be a CNF program with a valid¹⁴ set of annotations. Note that the dataflow annotations capture procedural notions such as intelligent backtracking, while still retaining a declarative nature. The set of all annotated CNF programs is termed the *execution space*. The declarative *abstract machine* for NRD programs is specified by the set of all possible executions — the execution space. That is, any implementation capable of realizing the entire execution space is a valid implementation of the abstract machine.

In particular, an interpreter for executing directly from the annotated CNF programs can be devised. This is evident since code generation was possible using only information local to each node. Therefore, we can modify the code generator proposed in section 3.4 to obtain an interpreter that actually implements the semantics instead of generating code. Such an interpreter is a faithful implementation of the abstract machine. Needless to say, the compilation approach has performance advantages.

In summary, we have proposed an abstract machine using annotated CNF programs. These programs retain the structure of the original program while encoding performance-critical aspects of the execution.

3.8 Relaxing the Tree Assumption

Up to this point, we have assumed the AND/OR graph generated from the CNF program is actually a tree. If the program contained common subexpressions (subtrees), we simply replicated the entire subexpression at each point where it was needed. We now present other techniques to deal with common subexpressions. In section 4, we show how these techniques can also be used to implement recursion.

We can eliminate common subexpressions by treating the subtree as a subroutine that is called from each occurrence. The subroutine analogy is useful; however, it is limited in that we need two return pointers instead of one (i.e., success and fail destinations) and in that the subroutine can return multiple solutions. Our implementation simply sets the return destinations and then jumps to the entry point of the subtree — no actual subroutines are used.

We need to be careful to prevent two invocations of this subtree from interfering with each other, for example, by changing the value of a variable or a relation cursor. One way of doing this is to guarantee that two calls to the subtree cannot be active at the same time. That is, each call to the subtree must compute all results before proceeding to the success point, i.e., use the LM or M execution methods.

Another approach is to assign each invocation its own environment. This is easily accomplished by declaring each variable in the subtree as a member of a structure and having each occurrence of the subtree declare its own local structure. We note that the environment contains not only the variables of the CNF program, but also all variables that store state information, such as *node_loc*, relations, cursors, etc. This allows us to pipeline tuples out of the subtree, resulting in P and LP executions.¹⁵ Since the number of

¹⁴The validity of an annotation is being used in an informal sense; for example, using preselect on a pipelined execution is invalid. Such annotations are being excluded.

¹⁵Separate environments can also be used with LM and M execution methods, however, this is unnecessary. In fact, the environment does not have to contain any state information for a node that is not being pipelined, since this node is already reentrant. Hence, the environment need only contain the state information up to a frontier of the subtree, delineated by LM or

occurrences of the subtree is known at compile-time, we can still allocate all the space for the program statically. However, variable references now involve a base-address plus displacement computation.

3.9 Implementation Notes

For the sake of convenience and brevity, we have chosen to limit our discussions in this paper to pure Datalog and, therefore, have not included complex terms. The actual \mathcal{LDL} implementation, however, does handle complex terms and the abstract machine outlined here can easily be modified as in the actual implementation to include them. Additional preselect and postselect annotations for AND nodes are used to handle unification in the presence of complex terms, and the postselect annotations for OR nodes are also extended. To process the annotations, capabilities for constructing and checking structures, as well as accessing structure arguments must be provided. These capabilities also provide the facilities needed for assigning to free variables in complex terms and constructing indices when complex terms are involved. Note that arguments within a complex term need to be adorned as discussed in section 2.

A few details relevant to the code generation implementation are worth noting. First of all, a symbol table must be constructed and maintained during the code generation process to facilitate the correct assignment of names to variables (global and temporary), constants (including those from the query form), complex terms, indices, and relations (temporary and base). The use of a symbol table also facilitates the generation of declarations and initializations for variables. Secondly, it is imperative that before entering into a full materialization, the values currently assigned to bound variables (if any) be saved into temporary variables so that the values are not lost during the materialization process. The correct values may then be restored to the variables upon completion of the materialization.

It is interesting to note that the name of the `node_loc` variable may be passed down from an OR node to its successor AND nodes during code generation, obviating the need for the node table described in section 3.4.1. This is the approach taken in the \mathcal{LDL} implementation. It requires non-local information (with respect to a single node) but is easy to incorporate (along with other node table information used in recursion) into the simple recursive AND/OR tree (graph) traversal algorithm used during code generation.

It is apparent from the code descriptions presented in this section that the generated code will contain many goto statements which could be condensed or folded to provide a more succinct and efficient program. In the \mathcal{LDL} implementation, a postprocessing pass was utilized to fold goto statements as appropriate.

Finally, we point out that the \mathcal{LDL} compiler does not actually construct a fully annotated CNF prior to code generation. The compiler extracts only those rules relevant to the current query form from the original \mathcal{LDL} program and constructs the corresponding AND/OR tree. The \mathcal{LDL} optimizer then chooses the CNF structure and encodes it directly in the AND/OR tree. After choosing the structure, the optimizer annotates the nodes of the tree to provide most of the access method annotations discussed using a greedy algorithm.

The simple, naive code generator outlined in this paper was based on the existence of a fully annotated CNF which provided all the information necessary to generate code locally with respect to each node in the AND/OR tree. The actual \mathcal{LDL} code generator, however, is quite sophisticated, essentially computing

M nodes.

argument adornments and providing the dataflow destinations, intelligent backtracking destinations, as well as some of the postselect and preselect annotations on the fly as the target language code is generated.

4 Recursive Datalog

We now extend our model to include recursive Datalog programs. Our goal is to show that the techniques presented so far are, in fact, sufficient to compile recursive queries.

4.1 Execution Method Annotations

We extend the AND/OR graph to include two new types of OR nodes: *clique-entry* and *clique* nodes. These nodes correspond to recursive predicate occurrences. A clique node corresponds to a predicate occurrence in the rule of a predicate from the same recursive clique. A clique entry node corresponds to all other recursive predicate occurrences, i.e. the initial point of invocation of a recursive clique. The successors of a clique-entry node are the rules for all predicates in the recursive clique. Note that this may include rules for predicates other than the invoked predicate, because of mutual recursion. Clique nodes have no successors. The distinction between clique and clique-entry nodes is easily motivated by the desire to have a finite, non-cyclic representation of the clique, which makes traversing the AND/OR graph much easier.

The main challenge in implementing recursion is to keep multiple invocations of the recursive predicate from interfering with each other. This is essentially the same problem solved in section 3.8 in the context of common subexpressions, so we would expect the same solutions to apply.¹⁶ Briefly, the solutions were to make the code reentrant by using a relation to store all results of the computation or a complex structure to keep a unique environment for each entry point into the subtree.

If we choose to use a relation, we end up with a semi-naive, fixpoint approach to recursion [Ba85]. Note the materialized relation avoids having multiple invocations of the recursive predicate occurrence active at the same time by replacing the recursive subquery with the tuples computed in the previous iteration. By assuming the overhead of duplicate elimination, we are guaranteed to terminate.

If we choose to use separate environments, we have to allocate space for these environments dynamically, since we cannot tell how many recursive calls will be made a priori. This allocation can be based on a stack.¹⁷ This approach does not incur the overhead of duplicate checking, but cannot guarantee termination.

Regardless of whether we choose the semi-naive or stack-based approach, the four execution methods are still applicable. That is, we can pipeline a semi-naive computation, lazy pipeline a stack-based computation, etc. The resulting combinations provide a rich mixture of execution strategies. However, note that if we pipeline a semi-naive computation, the clique node is not reentrant. If there is more than one entry point into this clique, we can simply treat it as a common subexpression, and any of the techniques in section 3.8 apply.

¹⁶Of course, unfolding a recursive clique into a tree is simply out of the question!

¹⁷This is essentially the way Prolog implements recursion.

4.2 Dataflow Annotations

We briefly discuss the dataflow annotations of clique nodes, clique-entry nodes and their immediate successors. It is not necessary to introduce new dataflow points, only to describe how the dataflow destinations are modified for each of the new types of nodes.

A clique-entry node is simply a new type of OR node, and its dataflow destinations are similar to those of the OR node. The successors to the clique-entry node are the rules of the recursive clique. We group the rules for each recursive predicate together, and assume that the first group contains the rules for the predicate associated with the clique-entry. We choose an arbitrary order for the rules in each group, so that the notion of first, next, and last rule in the group are defined. The entry destination of the clique-entry node is the entry point of the first rule in the first group. The dataflow destinations for each group of rules are identical to those of the AND node successors to an OR node. In particular, the fail destination of the last rule for each recursive predicate is the fail node of the parent clique-entry node. The fail destination of the clique-entry node is different. It is easier to understand in the context of semi-naive computation. After all tuples for a recursive predicate have been found, we must find all tuples for the next recursive predicate. Thus, the fail destination of the clique-entry node can be the entry point of the first rule for any of the recursive predicates. Of course, it can also be the regular fail destination for an OR node. The decision of whether to go to the regular fail destination or to one of the rule entry points can be viewed as the fixpoint check in the computation.

By definition, clique nodes have no successors. Conceptually, however, a clique node is the same as a clique-entry node. Hence, we would expect that its dataflow destinations are the same as for the corresponding clique-entry node. We note here that the fail destination of the last AND node for a given clique predicate is no longer simply the fail point of the clique-entry node, but the fail point of one of the clique-entry or clique nodes. Again, the specific node destination may have to be determined at run-time.

Figure 4 illustrates the dataflow points and destinations for a specific recursive clique, with two recursive predicates, `p1` and `p2`, and two rules for each predicate. We assume the clique-entry node refers to the predicate `p1`. The general case is easier to infer than to draw, so it is left up to the reader.

4.3 Code Generation

In this section, we describe how code can be generated to implement both semi-naive and stack-based recursive computations. We first mention some conventions necessary to compile recursive queries. Then, we describe the code generated for a pipelined execution, and briefly outline the changes necessary to realize lazy pipelined, lazy materialized, and materialized executions.

4.3.1 Conventions

First of all, we assume the adornment algorithm has taken care of stability transformations [SZ87]. This is necessary, since we compile each clique only once, so the adornment of the clique-entry node must be subsumed by the clique nodes. Note that it is not strictly necessary that the converse be true, since any

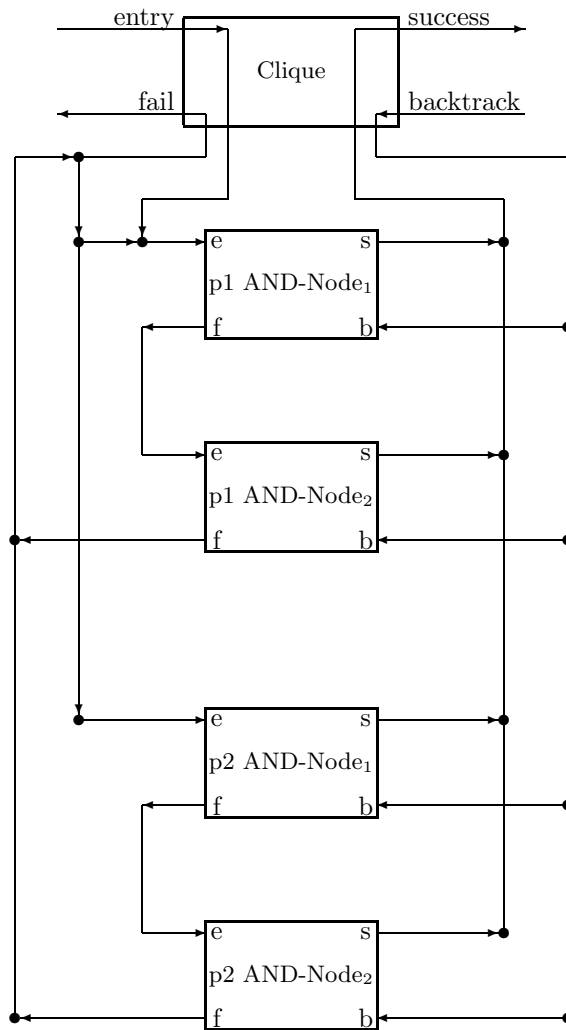


Figure 4: Dataflow Points and Destinations for Clique or Clique-Entry Node

extra bound arguments can be post-selected at the clique node.

For semi-naive computation, we assume that the rules have been rewritten into a form amenable for bottom-up processing. That is, magic-set, counting-set or some other rewriting transformation has taken place [Ba85,SZ86,BMSU86,BR87]. These rewriting techniques normally transform a recursive clique into two phases or fixpoint computations. We note that the first phase can be viewed simply as a clique embedded in the second phase. Hence, we need only discuss how code is generated for a single fixpoint.

In the semi-naive computation of a recursive clique, the clique-entry node must know which recursive predicate is currently being computed, in much the same way that an OR node needs to know the rule that is currently active. The name of the current predicate is stored in a variable called *node_pred*.

The semi-naive computation consists of repeatedly executing the recursive rules, where the results of each iteration are used in the subsequent pass. Thus, we need to manage the tuples generated at each iteration. For efficiency, we assume the *fact manager* (i.e., the underlying DBMS with a tuple-at-a-time interface) supports this model directly, by keeping the tuples generated in the current iteration separate from the entire relation. A cursor can then be declared as either a *delta* cursor, selecting only those tuples generated in the previous iteration, or a *total* cursor, which selects all tuples in the relation. The clique nodes in the recursive rules read the tuples in this *semi-naive* relation. For each clique node, we assume an annotation to determine whether a delta or total cursor should be used.¹⁸

In the stack-based approach, we use the stack, not the fact manager, to store temporary computations. This includes partial results, as well as the execution environment. Hence, all variable accesses must be resolved as offsets from a stack frame. The offsets can be computed easily if the stack frame is declared as a structure type, in which case the type name is sufficient to compute them. The type name of the stack frame used at each node can be found in the node table.

For each recursive invocation, we create a new environment in the stack. Since an environment remains active after a solution is found, the current environment is not necessarily at the top of the stack. Thus, we must assume two different pointers: one to the current environment and one to the top of the stack. Note that each environment must keep a pointer to its calling environment, since the two environments are not necessarily adjacent in the stack.

4.3.2 Semi-Naive Computation

In a pipelined execution, we would like to go to the success destination of the node every time we generate a tuple. Moreover, we do not wish to store the tuples generated in a relation — that is, at subsequent invocations we start work from scratch. In semi-naive recursion, however, it is necessary to store the tuples into a relation as they are generated. To be consistent with the pipelined approach, we must clear the relation after we generate all possible tuples. While this may seem strange at first, it is desirable in precisely the same cases where pipelining is. For instance, there is little duplication in the incoming binding pattern, and we wish to conserve memory resources.

We now describe the code generation for the clique and clique-entry nodes. As stated previously, the

¹⁸A total cursor is needed for non-linear recursion.

clique nodes are treated as base relations. It is only necessary to know the name of the relation being read and whether to open a delta or total cursor. This information is stored in the annotations for the clique node. Thus, the code for clique nodes is virtually identical to that for base relations in the non-recursive case.

A clique-entry node is a bit more complicated. Intuitively, the semi-naive execution consists of repeatedly “firing” each rule in the recursive predicate until no more tuples from it can be found. When all rules have been tried, we determine whether we have reached a fixpoint. If we have, then no more tuples can be generated for the clique, and so we go to the normal fail destination for the node. Otherwise, we proceed to “fire” each of the rules one more time.

Clearly, upon entry to the clique, there is no need to try any of the recursive rules in the clique, since these cannot possibly produce any tuples. Also, in the remaining iterations, it is unnecessary to try any of the non-recursive (exit) rules, since these cannot produce any new tuples. For brevity, we do not make this distinction in the following code description, but clearly this is something any implementation would use to its advantage (as does ours).

- **Clique-Entry:**

```

node_e: set node_pred to first clique predicate
        goto e_dest for node
node_b: goto node_loc; i.e., indirectly through this variable.
node_f: if node_pred is last clique predicate then
        indicate new semi-naive phase
        if last phase generated new tuples then
            set node_pred to first clique predicate
            goto entry point of first rule for first clique predicate
        else
            clear all clique relations
            goto f_dest for node
    else
        set node_pred to next clique predicate
        goto entry point of first rule for next clique predicate
node_s: construct and store a tuple in appropriate clique relation
        if node_pred is clique-entry predicate then
            if tuple matches input bindings then
                goto s_dest for node
    goto node_b

```

Two things should be noted. First of all, since there may be more than one clique predicate, at the success of the clique-entry node, we may have to store into more than one relation. We can determine the correct relation by looking at the value of *node_pred*. Secondly, before going to the success destination, we must determine whether the tuple just generated matches the input bindings. In the non-recursive case, this was unnecessary, because only matching tuples were generated. However, in the recursive case, this is not so. For example, to find the ancestors of *mary*, we may have to find the ancestors of *john* as well. However, we should not go to the success destination with one of *john*’s ancestors.

As in the non-recursive case, it is easy to modify the pipelining code to do lazy pipelining. The approach is to modify the entry point so that it can determine whether we are *done* computing the tuples for a

particular binding pattern. The backtrack point is modified so that it can either read from the materialized relation or proceed to the normal backtrack destination of the node. The success point stores any new tuples generated into the materialized relation, and the fail point marks the materialization for the current binding pattern as *done*. Since we are already storing tuples into a relation, these changes are easier to incorporate for the recursive case than for the non-recursive case. In fact, we simply have to avoid deleting the clique relations at the fail point. The tests for the *done* indicator are the same as in the non-recursive case.

As was the case with non-recursive queries, lazy pipelining can be converted into lazy materialization simply by changing the code for the success and fail points. In particular, at the success point we jump to the backtrack point rather than the success destination, and at the fail point we jump to the entry point rather than the fail destination.

Finally, the materialized execution can be viewed as a special case of lazy materialization, where no arguments are bound. For brevity, we do not include the code for the last three executions.

4.3.3 Stack-Based Computation

In a stack-based execution, there is no distinction between a clique-entry and a clique node. Essentially, they both consist of a subroutine call, where we pass parameters, set up a new execution environment, and receive some answers after the subroutine returns.

The main difficulty lies in setting up a new environment. When an environment is created, we need to set up the bound variables in this environment. That is, we must copy them from the previously active environment to the newly created one. Similarly, when an answer is found, the return variables must be copied to the environment of the caller. Finally, note the return address is also stored in the environment. Actually, two return addresses are required in our approach — the success and fail destinations for the current invocation.

As with the semi-naive computation, no changes are necessary to the AND nodes, so we omit them entirely in this description. We present the code for both clique-entry and clique nodes.

- **Clique or Clique-Entry:**

```

node_e: allocate space for new environment
         copy bound variables from previous to new environment
         set success destination of environment to be node_s
         set fail destination of environment to be node_f
         set frame pointer of environment to be old frame pointer
         set frame pointer to be new environment
         goto e_dest for node
node_b: set frame pointer to be new environment
         goto node_loc of environment; i.e., indirect
node_f: release space for new environment
         set frame pointer to be previous environment
         goto f_dest for node
node_s: copy results from new to previous environment
         set frame pointer to be previous environment
         goto s_dest for node

```

A few points from the above code should be noted. Since each recursive call produces only tuples that match the bound arguments, it is not necessary to check the tuples at the success point. This should be contrasted to the semi-naive case. Also, note that we only execute the rules for the predicate that is actually being invoked. This simplifies the code generated at the fail point considerably.

The transformations from pipelining to lazy pipelining, lazy materialization and materialization are the same as in the semi-naive case, and hence are not described here. However, some things should be noted. Recall that in a top-down execution there is no fundamental difference between a clique-entry node and a clique node. Thus, it is quite natural to achieve common subexpression elimination within the recursive calls. That is, after all the ancestors of `mary` have been discovered, there is no need to recompute them, even if the tuples are needed deep within the recursive computation. We expect this will lead to considerable savings in certain queries, for example, finding connected components of a graph.

4.4 Implementation Notes

As was the case for NRD, complex terms have been handled in the actual \mathcal{LDL} implementation and the model for recursive Datalog that is presented here can easily be modified simply using preselect and postselect annotations. The name of the stack frame may be passed down from the clique entry node to its successor AND nodes during code generation just as was possible with the backtrack location variable, again obviating the need for the node table. All other implementation details outlined for the NRD case also carry over for the recursive case. However, there are some points where recursion adds a level of complexity to the normal processing. These complications, as well as some necessary assumptions, are briefly discussed below.

Garbage collection is a traditional problem associated with recursion. In the semi-naive approach, the fact manager becomes responsible for garbage collection. This is accomplished when the clique relations are deleted. In the stack-based approach, on the other hand, the garbage collection problem is addressed when the stack is popped.

In section 3.6, we showed that intelligent backtracking can be implemented with no overhead for non-recursive queries. In the presence of recursion, intelligent backtracking becomes more complicated. Our experience with the \mathcal{LDL} implementation, however, has proven that intelligent backtracking can be implemented in the presence of recursion with little or no overhead. The solution outlined in section 3.6 carries over to the semi-naive approach with no changes. It is sufficient to note that when we are pipelining, we must clear the clique relations at the entry point, not the fail point, of the clique, since the fail point may be bypassed during intelligent backtracking. In the stack-based approach, however, we must clean up the stack after backtracking over a clique. At first, this may seem like unnecessary overhead, but the amount of work is little more than that involved in cleaning up the stack without intelligent backtracking.

In this section we assumed that the fact manager supports the semi-naive model directly. This required adding a fact manager routine to declare when we enter a new phase of the computation — that is, a new iteration in our fixpoint computation. In return, the fact manager supports delta cursors, where only the tuples generated in the previous phase are returned. We chose to add these concepts to the fact manager, rather than to emulate them at the inference engine level, purely for efficiency. In fact, we store tuples in such a way that only a “high-water mark” is necessary to implement a semi-naive scan.

Essential to our ability to extend the Abstract Machine to handle recursive Datalog is the existence of two preprocessing phases, one which analyzes the rules and groups all strongly connected components into recursive cliques and one which generates the annotated CNF for the recursive cliques. In the latter processing phase, non-trivial transformations must take place to map the program into an equivalent one where, for the semi-naive case, constants are pushed through the recursion [BMSU86, BR87, SZ86]. Both processing phases exist as part of the current \mathcal{LDL} implementation.

5 Conclusions

In this paper, we proposed an abstract machine for \mathcal{LDL} that maintains a high-level view of an \mathcal{LDL} program while incorporating aspects of its execution that make a performance difference. The abstract machine is defined by the execution space, i.e., the set of all annotated CNF programs, each of which models an actual execution. The CNF program provides the skeleton of the execution, and the annotations specify other relevant details, such as access methods, join methods, execution strategies, intelligent backtracking, etc. We formalized four execution methods (top-down, bottom-up, as well as two hybrid methods that incorporate memoing) and two recursive computations (fixpoint and stack-based). The two computations and four execution methods were shown to provide a rich variety of recursive techniques.

We presented a realization of this abstract machine by describing the code generation algorithm used in the \mathcal{LDL} compiler. This algorithm proceeds by translating each node in the annotated CNF program into a sequence of imperative statements, including calls to a tuple-level interface of an underlying DBMS. In this paper, we presented the abstract machine in the context of Datalog. The experience in implementing the compiler for \mathcal{LDL} has shown that this abstract machine is sufficient to support advanced features, such as sets, negation, updates and non-deterministic choice.

There are many aspects of execution that have not been explicitly addressed in the above proposal. The use of parallelism and peep-hole optimization, for example, are of special interest to us. Further, the abstract machine provides a framework for comparison of various techniques proposed in the literature. For example, many recursive query processing techniques can be modeled and compared using this abstract machine.

Acknowledgements

We are grateful to Kevin Greene for careful reading of an earlier version of this paper. We are also thankful to Manuel Hermenegildo, Roger Nasr and Carlo Zaniolo for providing a forum for preliminary discussions leading to this abstract machine.

References

- [Ba85] Bancilhon, F.D. "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems*, edited by M. Brodie and J. Mylopoulos, Springer-Verlag, 1985.

- [BMSU86] Bancilhon, F.D., D. Maier, Y. Sagiv, and J. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs," in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, 1986.
- [BR86] Bancilhon, F.D. and R. Ramakrishnan. "An Amateur's Introduction to Recursive Query Processing Strategies," in *Proc. SIGACT-SIGMOD Int. Conf. on Management of Data (SIGMOD)*, Washington D.C., 1986.
- [BR87] Bancilhon, F.D. and R. Ramakrishnan. "On the Power of Magic," in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, 1987.
- [By80] Byrd, L.. "Understanding the Control Flow of Prolog Programs," in *Proc. of the Logic Programming Workshop*, 1980.
- [HU79] Hopcroft, J. and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.
- [KZ88] Krishnamurthy, R., and C. Zaniolo. "Optimization in a Logic Based Language for Knowledge and Data Intensive Applications," *Extending Data Base Technology*, Venice, 1988.
- [Ll84] Lloyd, J.W. *Foundations of Logic Programming*, Springer Verlag, 1984.
- [Me82] Mellish, C. "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter," in *Logic Programming*, edited by K.L. Clark and S.A. Tärnlund, Academic Press, 1981.
- [Mi68] Michie, D. "'Memo' Functions and Machine Learning," in *Nature*, April 1968.
- [Nil80] Nilson, N.J. *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.
- [NT89] Naqvi, S. A. and S. Tsur. *A Language for Data and Knowledge Bases*, W.H. Freeman, 1989.
- [PP82] Pereira, L.M. and A. Porto. "Selective Backtracking," in *Logic Programming*, edited by K.L. Clark and S.A. Tärnlund, Academic Press, 1981.
- [RBK88] Ramakrishnan, R., C. Beeri, and R. Krishnamurthy. "Optimizing Existential Queries," in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, Austin, April 1988.
- [SZ86] Sacca, D. and C. Zaniolo. "The Generalized Counting Method for Recursive Logic Queries," in *Proc. 1st Int. Conf. on Database Theory*, Rome, 1986.
- [SZ87] Sacca, D. and C. Zaniolo. "Implementation of Recursive Queries for a Data Language Based on Pure Horn Logic," in *Proc. 4th Int. Conf. on Logic Programming*, Melbourne, 1987.
- [Ull85] Ullman, J. "Implementation of Logical Query Languages for Databases," in *TODS*, Vol. 10, No. 3, pp. 289-321, 1985.
- [Ull88] Ullman, J. *Database and Knowledge Systems, Vol. I*, Computer Science Press, 1988.
- [Ull89] Ullman, J. *Database and Knowledge Systems, Vol. II*, Computer Science Press, 1988.
- [Vi89] Vielle, L. "Recursive Query Processing: The Power of Logic," to appear in *Theoretical Computer Science*, 1989.
- [Wa71] Wadsworth, C.P.. "Semantics and Pragmatics of the Lambda-Calculus," D. Phil. Thesis, University of Oxford, 1971.
- [Wa83] Warren, D.H.D. "An Abstract Prolog Instruction Set," Technical Note 309, SRI International, 1983