Incremental, Bottom-Up, Well-Founded Deduction

Brian W. Beach
Software and Systems Laboratory
HPL-91-20
February, 1991

expert systems; KBMS; DATALOG; incremental computation; logic programming

In this paper we describe an algorithm, called AFP, for the bottom-up execution of logic programs well suited to data-driven applications, where incremental updates to the base relations are propagated through the program to produce updated answers without completely re-executing the program. We show that AFP conforms to the well-founded semantics for general logic programs and that it terminates in polynomial time for DATALOG programs with negation.

AFP is based on a modified formulation of well-founded negation that does not rely on computing sets of false atoms, avoiding the instantiation of the entire Herbrand base. AFP solves the two major problems in incremental bottom-up computation, recursion through negation and self-supporting conclusions, by using a counting technique over a two-dimensional representation of time to detect and stop such loops. Its feasibility has been verified with an implementation in Prolog.

Internal Accession Date Only

1 Introduction

A general logic program is one that includes both positive and negative subgoals in the body of a rule. We follow the convention expounded by Reiter [Rei78] of viewing a program as having two disjoint components: the *extensional database* (EDB), that we will call the *facts*, and the *intensional database* (IDB), that we will call the *rules*. In data-driven applications, the set of facts varies over time, and the consequently changing conclusions of the rules are used to trigger actions in the application.

There are two fundamentally different approaches to executing a logic program: bottom-up and top-down. In bottom-up computation [Ull89], one starts with the facts constituting the EDB, matches them against the premises in the rules in the IDB, and for all rules where the body is known to be true, concludes that the head of the rule is true. Conclusions drawn are then fed back into the process and used as the basis for further conclusions. This process continues until no more conclusions can be drawn, i.e. when the fixed point has been reached. Top-down evaluation in the context of the well-founded semantics has been studied by Ross [Ros89].

In top-down computation a goal (query) is presented for solution. All rules are found whose heads match the goal. The literals in the bodies of these rules are then taken as subgoals to be proved in the same manner. The process stops when a subgoal is found that is in the EDB or that does not match the head of any rule.

There are advantages and disadvantages to both approaches. Bottom-up evaluation can produce unneeded answers, while top-down evaluation can make many attempts before finding an avenue that leads to a proof. Despite its drawbacks, bottom-up computation is useful in situations where the EDB is changing. In such situations incremental update techniques can greatly improve performance; a small change to the EDB can make small incremental changes to the set of conclusions without having to recompute them all.

One source of inspiration for AFP has been the work on production systems. Production systems require that incremental updates to the EDB be efficiently processed to generate a new set of active rules. The RETE algorithm developed by Forgy [For79, For81], and the TREAT algorithm developed by Miranker [Mir87, NGR88, MJF+90] are examples of incremental, bottom-up evaluators.

Our second source of inspiration comes from Van Gelder's alternating fixpoint definition of the semantics of general logic programs [VGRS88, PP90]. Negation through recursion can cause RETE-style algorithms to oscillate between two answers. The alternating fixpoint solves this program by defining a middle ground for undefined atoms.

A problem with prior incremental algorithms is that once concluded, an atom may become self supporting and later changes in the EDB can leave conclusions present without any support in the EDB. The counting technique AFP uses to stop recursion solves this

problem.

AFP solves these two major problems in incremental bottom-up computation: recursion through negation and self-supporting conclusions. An implementation in Prolog has been written to confirm its viability.

The rest of this paper is organized as follows: Section 2 describes our formulation of the alternating fixpoint; Section 3 describes the normal form that we use to ensure that programs can be easily translated into the rule/goal networks of Section 4. Sections 5 and 6 are the core of the paper and describe the data represtation and actions of the AFP algorithm. In Sections 7 and 8 we talk about some optimizations that can be made to AFP both for general programs and for stratified programs. Finally in Section 9 we conclude with pointers to future work in this area.

2 The Alternating Fixpoint

An alternating fixpoint computation is a sequence of estimates of the set of atoms that are true, alternating between overestimates and underestimates and converging on the correct answer. Each successive overestimate is a smaller set, excluding more atoms known to be false. Each successive underestimate is a larger set, including more atoms known to be true. An overestimate is used to construct the next underestimate, and vice versa. It has been shown that these estimates will converge to a fixpoint where the underestimate includes all true atoms, the overestimate excludes all false atoms, and the remaining atoms are undefined [VG89].

Each of the steps in this alternating fixpoint is itself a fixpoint computation that computes a model for the program based on the previous estimate. We call each of these an *inner fixpoint* and the computation as a whole the *outer fixpoint.*

The key to the alternating fixpoint is that the value of a negated subgoal is based on the previous iteration of the outer fixed point computation. For a program including only the rule $p \leftarrow \neg p$ the first guess is that p is true. The first iteration uses this assumption to determine the status of $\neg p$, concluding that p is false. The second iteration uses the previous conclusion to determine that p is true. The computation will alternate between these two states. Since this is the fixed point, we know that p is not true because it is not in the underestimates, and that p is not false because it is in the overestimates. Therefore the value of p must be undefined, which is correct according the well-founded semantics.

In the following discussion we use H to denote the Herbrand base of the program, containing all atoms built from the predicates and constants in P, and P_H to denote the Herbrand instantiation of the program, consisting of all possible instantiations of the rules in P using H.

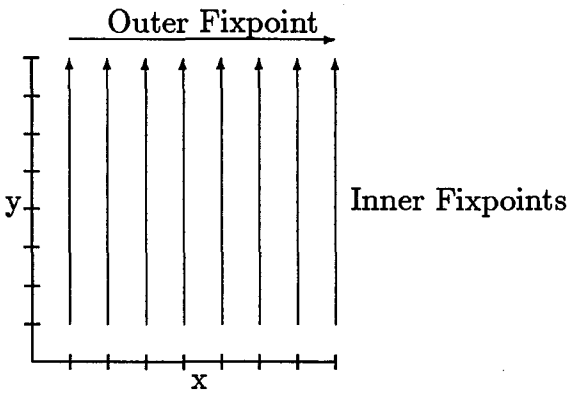

Figure 1: Representation of Time

Definition 1: $I(A, B)$, the *immediate consequence* function of a program P, gives the set of atoms P can derive using A for positive subgoals and B for negative subgoals.

$$I(A,B) = \{p \in H \mid \exists(p \leftarrow q_1 \ldots q_n, \neg r_1, \ldots, \neg r_m) \in P_H, \text{ with every } q_i \in A, \text{and every } r_i \notin B\}$$

We use a two-dimensional notion of *time* to describe the progression of the alternating fixpoint. The inner fixpoint progresses in the positive y direction (upward) and the outer fixpoint progresses in the positive x direction (to the right), as shown in Figure 1. Time (x_1, y_1) is *before* time (x_2, y_2) if $x_1 < x_2$ or $x_1 = x_2$ and $y_1 < y_2$. A time (x, ∞) denotes the result of one inner fixpoint.

The overestimate at time (x, y) is denoted by $O_{x,y}$, and the underestimate at time (x, y) is denoted by $U_{x,y}$. The equations below give the contents of each set. Each inner fixpoint starts with E, the contents of the EDB and draws all possible conclusions from it.

$$\begin{array}{lcll} O_{0,\infty} & = & H & \\ O_{x,0} & = & E & (x > 0) \\ O_{x,y} & = & O_{x,y-1} \cup I(O_{x,y-1}, U_{x-1,\infty}) & (x > 0 \text{ and } y > 0) \\ U_{x,0} & = & E & \\ U_{x,y} & = & U_{x,y-1} \cup I(U_{x,y-1}, O_{x,\infty}) & (y > 0) \end{array}$$

The outer fixpoint proceeds: $O_{0,\infty}$, $U_{0,\infty}$, $O_{1,\infty}$, $U_{1,\infty}$, $O_{2,\infty}$, $U_{2,\infty}$, et cetera.

The inner fixpoint computations are monotonically non-decreasing positive induction, because the negated conditions depend on a set held constant. Thus, $U_{x,y} \subseteq U_{x,y+1}$ and $O_{x,y} \subseteq O_{x,y+1}$. Van Gelder has shown [VG89] that the outer fixpoint has non-decreasing underestimates and non-increasing overestimates: $U_{x,y} \subseteq U_{x+1,y}$ and $O_{x,y} \supseteq O_{x+1,y}$.

3 Normal Form

We make three basic assumptions about the structure of programs to be interpreted by AFP. First, that they are bottom-up evaluable, second that they are in normal form, and that they are function-free (i.e. DATALOG).

A logic program is bottom-up evaluable if, for every rule, every variable in the head of the rule is contained in the body of the rule [BR88]. Every conclusion drawn by the program will be fully bound.

We can assume, without loss of generality, that the set of predicates in the EDB and the set of predicates in the IDB are disjoint. A program is said to be in normal form if every predicate in the IDB has exactly one rule that can deduce it, and that every rule is in one of the three forms shown below. We use $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ to denote vectors of arguments, and $\mathbf{v}(\mathbf{X})$ to denote the set of variables in $\mathbf{X}$. Because we require that a program be bottom-up evaluable, every variable in the head of a rule must occur in the body and every variable in a negated subgoal must also appear in a positive subgoal. The necessary restrictions on the sets are shown next to each type of rule.

$$\begin{array}{lcll} p(\mathbf{X}) & \leftarrow & q(\mathbf{Y}) \vee r(\mathbf{Z}) & \text{where } \mathbf{v}(\mathbf{X}) \subseteq \mathbf{v}(\mathbf{Y}) \text{ and } \mathbf{v}(\mathbf{X}) \subseteq \mathbf{v}(\mathbf{Z}) \\ p(\mathbf{X}) & \leftarrow & q(\mathbf{Y}) \wedge r(\mathbf{Z}) & \text{where } \mathbf{v}(\mathbf{X}) \subseteq \mathbf{v}(\mathbf{Y}) \cup \mathbf{v}(\mathbf{Z}) \\ p(\mathbf{X}) & \leftarrow & q(\mathbf{Y}) \wedge \neg r(\mathbf{Z}) & \text{where } \mathbf{v}(\mathbf{X}) \subseteq \mathbf{v}(\mathbf{Y}) \text{ and } \mathbf{v}(\mathbf{Z}) \subseteq \mathbf{v}(\mathbf{Y}) \end{array}$$

Note that the first form is not allowed in logic programs, but is equivalent to the two rules $p \leftarrow q$ and $p \leftarrow r$. Condensing them into one rule will make the AFP algorithm description below much simpler.

An arbitrary general logic program can be transformed into a normalized program using three rewriting rules. The first rewriting rule is used to produce a unique rule that can conclude each predicate. If there are two rules to conclude the same predicate p, then they are rewritten into three rules: one disjunctive rule, and two rules concluding new predicates corresponding to the initial two rules. Two rules of the form:

$$\begin{array}{rcl} p(\mathbf{X}_a) & \leftarrow & q_1(\mathbf{Y}_1) \wedge \ldots \wedge q_n(\mathbf{Y}_n) \\ p(\mathbf{X}_b) & \leftarrow & r_1(\mathbf{Y}_1) \wedge \ldots \wedge r_m(\mathbf{Y}_m) \end{array}$$

become three rules:

$$\begin{array}{rcl} p(\mathbf{X}) & \leftarrow & p_a(\mathbf{X}) \vee p_b(\mathbf{X}) \\ p_a(\mathbf{X}_a) & \leftarrow & q_1(\mathbf{Y}_1) \wedge \ldots \wedge q_n(\mathbf{Y}_n) \\ p_b(\mathbf{X}_b) & \leftarrow & r_1(\mathbf{Y}_1) \wedge \ldots \wedge r_m(\mathbf{Y}_m) \end{array}$$

where $\mathbf{X}$ is a vector of n variables (p takes n arguments), and p_a and p_b are both new predicate names not used in the program.

The second rewrite rule is use for rules with a single subgoal in their body, or with no positive subgoals. A rule of the form $p(\mathbf{X}) \leftarrow q(\mathbf{Y})$ becomes $p(\mathbf{X}) \leftarrow t \wedge q(\mathbf{Y})$, and a rule of the form $p(\mathbf{X}) \leftarrow \neg q_1(\mathbf{Y}_1) \wedge \ldots \wedge \neg q_n(\mathbf{Y}_n)$ becomes $p(\mathbf{X}) \leftarrow t \wedge \neg q_1(\mathbf{Y}_1) \wedge \ldots \wedge \neg q_n(\mathbf{Y}_n)$, where t is a special EDB predicate that is always true.

The third rewrite rule is used for rules with more than two subgoals; it takes two positive subgoals, or a positive subgoal and a negative subgoal and makes a new rule with them, substituting the new predicate back in the original rule. If we start with a rule

$$p(\mathbf{X}) \leftarrow q_1(\mathbf{Y}_1) \wedge q_2(\mathbf{Y}_2) \wedge \ldots \wedge q_n(\mathbf{Y}_n)$$

we replace it with two rules:

$$\begin{aligned} p(\mathbf{X}) &\leftarrow p'(\mathbf{Z}) \wedge q_3(\mathbf{Y}_3) \wedge \ldots \wedge q_n(\mathbf{Y}_n) \\ p'(\mathbf{Z}) &\leftarrow q_1(\mathbf{Y}_1) \wedge q_2(\mathbf{Y}_2) \end{aligned}$$

$$\text{where } \mathbf{v}(\mathbf{Z}) = (\mathbf{v}(\mathbf{Y}_1) \cup \mathbf{v}(\mathbf{Y}_2)) \cap (\mathbf{v}(\mathbf{X}) \cup \mathbf{v}(\mathbf{Y}_3) \cup \ldots \cup \mathbf{v}(\mathbf{Y}_n))$$

and p' is a new predicate name not appearing elsewhere in the program. $\mathbf{Z}$ is a vector of all variables in $\mathbf{Y}_1$ and $\mathbf{Y}_2$ that are used elsewhere in the rule.

Lemma 3.1: A program translated to normal form has the same meaning under the well-founded semantics as the original program.

Positive equality in a rule can, via syntactic transformations, be eliminated. Negative equality can be handled as a filter on the atoms in the bottom-up computation [VGT89]. We will ignore the issue of equality in the rest of the paper.

The example we will use throughout this paper is Ross's program to find numbers with odd numbers of prime factors [Ros90], shown below [1].

$$\begin{aligned} p(X) &\leftarrow b(X) \\ p(X) &\leftarrow e(X, Y, Z) \wedge \neg p(Y) \wedge p(Z) \end{aligned}$$

The normal form is shown below.

$$\begin{aligned} p(X) &\leftarrow p1(X) \vee p2(X) \\ p1(X) &\leftarrow t \wedge b(X) \\ p2(X) &\leftarrow p3(X, Z) \wedge p(Z) \\ p3(X, Z) &\leftarrow e(X, Y, Z) \wedge \neg p(Y). \end{aligned}$$

[1] b and e are EDB predicates: $b(X)$ is true if X is prime and $e(X, Y, Z)$ is true if X has factors Y and Z.

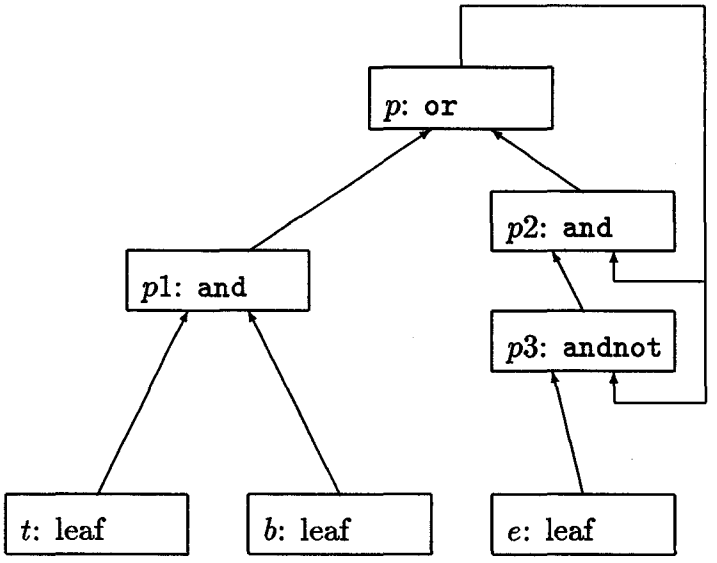

Figure 2: Network of nodes for prime number program.

4 Translation from rules to network

Every predicate in the program corresponds to one node in the network. EDB predicates correspond to *leaf* nodes and rule predicates correspond to *rule* nodes. Each EDB predicate has a single leaf node that sends any changes to that predicate to the rule nodes depending on it.

Since each IDB predicate in a normal-form program has exactly one rule, we can establish a one-to-one-to-one correspondence between predicates, rules, and nodes. There is a different type of rule node for each of the three rule types in normal form:

1. `or` nodes for disjunctive rules,
2. `and` nodes for conjunctive rules without negation, and
3. `andnot` nodes for conjunctive rules with negation.

The two inputs to a rule node are connected to the two nodes corresponding to the predicates it depends on. The output of a rule node is connected to all rule nodes depending on it.

The network for the prime number program is shown in Figure 2.

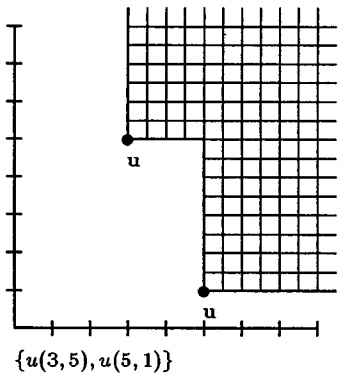

Figure 3: Sample shape of U

5 Representation of Fixpoint Stages

An alternating fixpoint computation involves the creation of an infinite number of sets $O_{x,y}$ and $U_{x,y}$. Obviously, we can't represent all of these sets explicitly.

Let's examine a few properties of O and U. For a given atom p, the set of (x, y) values where p in in $O_{x,y}$ and $U_{x,y}$ can be thought of as an area in the first quadrant including all of the integral points (x, y) where p is a member of the set. The set of possible shapes for these areas is highly restricted. We know that U is monotonically nondecreasing in the positive x and y directions; thus, if $p \in U_{x,y}$ then $\forall x' > x, y' > y: p \in U_{x',y'}$. The shape of the resulting area is as shown in Figure 3, and can be represented by the set of points at the lower-left corners.

The shapes for O are symmetrical because O is nondecreasing with y and nonincreasing with x; they can be defined by the positions of the lower-right corners. Since an atom may be present in all overestimates, the last lower right corner may be at $x = \infty$.

We do not, however, want to do this. In order to show termination later we will need to know that the algorithm proceeds only forward in time. As the computation progresses it will start at the left of the diagram and work to the right, and will need to know about the presense of an atom p in O before it gets to the lower-right corner. To accommodate these requirements we will represent the area redundantly using both lower-left and lower-right corners. A lower-left corner indicates that p is present in O, and a lower-right corner indicates that p is no longer present. The lower-right corner is in the first outer iteration where p does not appear. See Figure 4.

We use *tokens* to represent the points on the area charts. There are three types of tokens: u, l, and r. A token of the form $u(p, x, y)$ represents an underestimate that becomes true at time (x,y). A token of the form $l(p, x_1, y)$ represents the time that atom p first enters

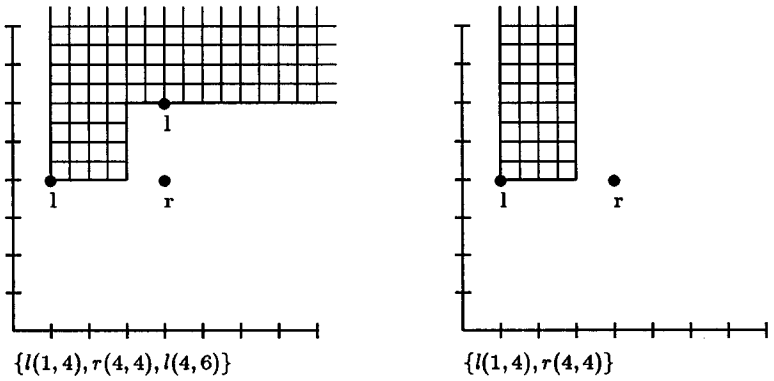

Figure 4: Sample shapes of O

the overestimate set, or steps up to a higher y level. Each lower right corner is represented by a token of the form $r(p, x_2, y)$, matching some l token with the same y.

More precisely, the equation defining the set T of tokens corresponding to a given O and U is shown below.

$$\begin{aligned} T &= \{u(p,x,y) \mid p \in U_{x,y}, p \notin U_{x-1,y}, p \notin U_{x,y-1}\} \\ &\cup \{l(p,1,y) \mid p \in O_{1,y}, p \notin O_{1,y-1}\} \\ &\cup \{r(p,x,y) \mid p \in O_{x-1,y}, p \notin O_{x,y}, p \notin O_{x-1,y-1}\} \\ &\cup \{l(p,x,y) \mid x > 1, p \in O_{x,y}, p \in O_{x-1,y-1}, p \notin O_{x,y-1}\} \end{aligned}$$

The most important property of this representation is that the set of tokens representing the contents of U and O up to a time (x, y) depends solely on the contents of U and O up to (x, y), and do not depend on their later contents. This allows the algorithm to proceed strictly forward in time. Once the tokens up to time (x, y) have been established, they will never have to be modified. This is why the tokens representing the lower-right corners of overestimates are one unit to the right of the actual corner.

Theorem 5.1: Changing U or O at time (x, y) will not change the set of token in T at times before (x, y).

Proof: This follows directly from the definition of T. Every membership in T of a token at time (x, y) is predicated only on the contents of U and O at earlier times. ∎

It is also important that we have a finite representation of U and O for atoms whose status is *unknown*, i.e. those that are present in all $O_{x,\infty}$ but not in any $U_{x,\infty}$. This is the reason that overestimates and underestimates are represented separately.

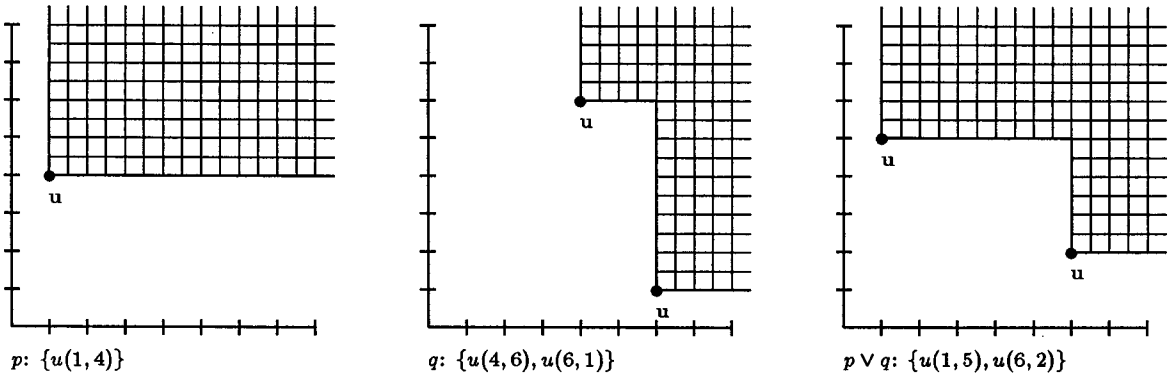

Figure 5: Disjunction of two underestimates

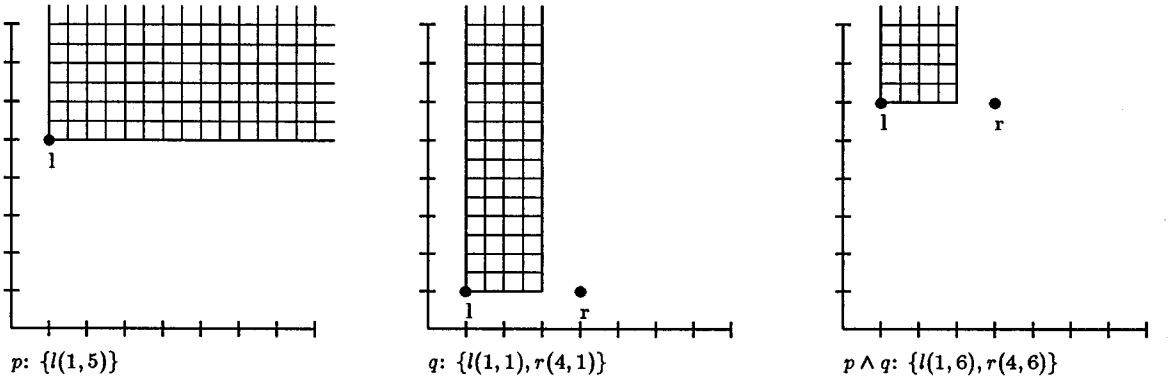

Figure 6: Conjunction of two overestimates

A disjunctive rule of the form $p \leftarrow q \vee r$ will place p in all U where either q or r is present at the previous time and in every O where either q or r is present at the previous time. Figure 5 shows the disjunction of two underestimates.

Conjunctive rules such as $p \leftarrow q \wedge r$ will place p in every set $U_{x,y}$ and $O_{x,y}$ where p and q both appear in the preceding U or O of the inner fixpoint. Figure 6 shows the conjunction of two overestimates. With both conjunctive and disjunctive rules, both inputs are always either underestimates or overestimates; the two are never mixed. This is because the inputs and the output are all within the same inner fixpoint.

5.1 Negation

Conjunctive rules with negation of the form $p \leftarrow q \wedge \neg r$ are a little different. Recall that the immediate consequence function I treats negation specially by looking for negated subgoals in the previous iteration of the outer fixpoint. Let's consider the underestimates

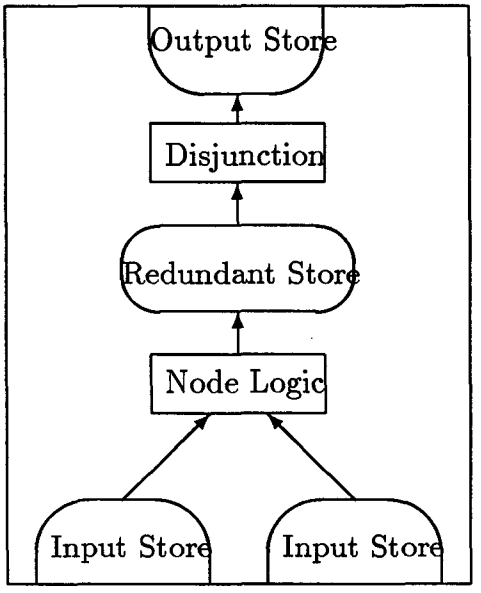

Figure 7: The internal structure of a node

and overestimates separately. For any q that is an underestimate, the corresponding r will be an overestimate. All atoms are present in $O_{0,\infty}$, so a conjunctive rule with negation can never conclude that any atoms are in $U_{0,y}$. The presence of an overestimate r that is in all overestimates until $O_{x,\infty}$ will prevent p from being in any underestimates until $U_{x+1,y}$. In other words, the fact that r is true until outer iteration x will prevent p from being in any underestimates until time $x+1$. The reverse case where q is an overestimate and r is an underestimate is symmetrical. The fact that r first appears in $U_{x,y}$ will prevent p from being in any overestimates beyond $O_{x,y}$.

6 The AFP Algorithm

Internally, each node consists of two input stores, some type-dependent logic, a redundant store, disjunction logic, and an output store, as shown in Figure 7. The input stores hold the tokens that have come from other nodes. The logic keeps the redundant store up to date based on the input, although because some parts of the input relations may be projected out the results may contain redundant tokens. This redundancy is removed by the disjunction logic, and the minimal set of output tokens to represent the results are put into the output store.

The storage of the input sets is necessary so that incremental joins can be done. When a token is added to or removed from one input set, the change must be compared with the contents of the other input set to determine what change to the output is required.

A *change* is the addition or removal of a token. We will denote these two types of changes using + and −. The addition of an underestimate is denoted by $+u(p, x, y)$, while its removal is denoted by $-u(p, x, y)$.

Every change to the EDB generates two changes that are propagated through the network. The addition of atom p generates $+u(p, 0, 0)$ and $+l(p, 1, 0))$. The removal of atom p generates $-u(p, 0, 0)$ and $-l(p, 1, 0)$.

Changes produced by one node must be forwarded on to the nodes it is connected to, but to ensure correctness, we must make sure that the computation proceeds forward in time. Nodes must not be allowed to produce results until their inputs are certain. All of the pending changes are stored in a list, sorted by time. At each step in the processing, the change with the lowest time is removed from the list and processed, producing further changes. As we will see shortly, these further changes will have higher times than the original.

Each type of node is responsible for ensuring that its set of output tokens is consistent with its input set. This is important, because it is the basis for the proof of correctness for AFP. Whenever a node receives a change to one input set, it calculates the corresponding changes to its output set and puts them into the sorted list. Incremental changes are handled by keeping the tokens in each store sorted by atom, and within each atom sorted by time.

Changes to the redundant store are processed by the disjunction logic. When a token is added, it is inserted into the sorted list. For underestimates, there are two cases to handle: (1) When the new token covers an area that is a subset of that covered by some token already present, no changes to the output store are needed, and (2) when the new token is does not cover a subset it is sent to the output store and any tokens previously there that are now redundant are removed. The removal of a token reverses the process. For overestimates, each pairs of l and r tokens is treated as a unit inside the node. A similar algorithm comparing the areas covered is used to update the output store.

Changes to the input stores are handled by the node-specific logic. In `or` nodes, each input change is forwarded directly to the redundant store and the disjunction logic takes care of it. Incremental joins are done by the logic in **and** nodes. Every newly changed area on one side is intersected with all areas for matching atoms on the right side and the results sent to the redundant store. The specific comparison done between the atoms is defined by the rule that the node corresponds to. For the rule $p(X, Z) \leftarrow q(X, Y) \wedge r(Y, Z)$, the second argument to the predicate on the left is matched against the first argument to the predicate on the right.

An `andnot` node must deal with a time shift. Tokens from the positive side are compared against tokens from the negated side at the previous x time. Because of this time shift, of the two areas being compared one will be an overestimate and the other will be an

underestimate. The y time coordinates from the negated side can be ignored, because the inference rule looks only at the final result of the inner fixpoint. For each atom p, an andnot node keeps track of the highest x where p is present in the overestimate on the negated side, and the lowest x where p is present in the underestimate on the negated side. These x values are used to trim the areas of the matching tokens on the positive side.

To summarize, the AFP algorithm proceeds as follows.

1. For each addition of an atom p to the EDB, create two changes and put them in the sorted list: $+u(p,0,0)$ and $+l(p,1,0)$.
2. For each removal of an atom p from the EDB, create two changes and put them in the sorted list: $-u(p,0,0)$ and $-l(p,1,0)$.
3. While there are changes in the list, take out the one with the lowest time and process it through all nodes taking it as input, generating more changes to add to the list. If there are ever two changes in the list that cancel each other out, such as $+u(p,x,y)$ and $-u(p,x,y)$, remove both of them.

Theorem 6.1: Processing a token through a node will only produce tokens with later times than the input token.

Proof: By definition, the contents of U and O at time (x,y) depend only on the contents of U and O at prior times. This means that changing U or O at time (x,y) can only affect U or O at later times. We know by Theorem 5.1 that this will affect only those tokens at times after (x,y). ∎

Theorem 6.2: AFP precisely models the alternating fixpoint. The U and O defined by the tokens produced by AFP are exactly the same as the U and O defined by the alternating fixpoint.

Proof: The proof is by induction on time; we show that after AFP terminates, the correct answer is produced at every time (x,y). The base case is $(0,0)$, at which time the correct state is stored in the leaf nodes. No other node can produce any tokens affecting time $(0,0)$, because nodes always produce tokens at times later than the times of input tokens. Assume that the correct state exists for all times $t_1 < t$. The output of every node will be correct through time t, because the only tokens that can affect time t are from previous times, and those are all correct. ∎

Theorem 6.3: AFP will terminate in polynomial time (in number of token comparisons) for DATALOG programs with negation.

Proof: We know that the alternating fixpoint will reach a fixpoint at some finite x, bounded by a polynomial. Each inner fixpoint is also polynomial bounded, and the number of possible tokens at each (x, y) is bounded by $|H|$, the size of the Herbrand base, which is bounded by a polynomial. The time spent processing each token is also bounded by $|H|$. The product of all of these is a polynomial. ∎

Theorem 6.4: Each removal processed represents a corner in the old state that is not present in the new state, and each addition processed represents a corner in the new state that is not present in the old state. Each change processed represents the number of changes processed is the same as the number of corners on the area diagrams.

Proof: Because AFP accurately models the alternating fixpoint, and because there is a unique representation of U and O using tokens, and because AFP proceeds forward in time, when a token is processed it must be necessary to represent the contents of U or O. ∎

Corollary 6.5: The number of tokens processed during an incremental change is bounded by the sum of the number of tokens needed to represent the old U and O and the number of tokens needed to represent the new U and O.

7 Stratified Programs

A stratified program is one that has no recursion through negation [Ros90]. In a stratified program, all of the answers can be computed in one stratum before going on to the next. This means that once in a stratum, the incoming changes can be considered to be EDB updates, and the times on the tokens can be reset to 0. This will save time by avoiding the production of l tokens that are sure to have matching r tokens.

To implement this approach, one would generate a rule dependency graph for the program and find strongly connected components (SCCs), label each SCC with a number greater than that for any SCC providing input to it, and include this number in the time attached to a token so that the time is now a triple: (s, x, y), where s is the stratification level. Now, when a token passes into a new stratum, the s value is set to that of the new stratum, and the x and y values are reset. This can be done by adding a `stratum` node to every link between strata.

The addition of the stratum number to the time is necessary so that all processing will be done in lower strata before working on upper strata.

In stratified programs, the outer fixpoint will be reached after exactly two inner fixpoints: one for underestimates and one for overestimates, thus exactly two tokens will be produced for each conclusion reached by the program.

8 Optimization

In an efficient implementation it would be possible to have only one token store for each predicate, used for the redundant store inside the node, the output store for the node and the input stores of the dependent nodes.

A further optimization can be achieved by not differentiating between overestimates and underestimates in strongly connected components where there is no recursion involving negation, and where all inputs come from similarly constrained strongly connected components.

Also, disjunctive nodes will work with more than two inputs. To reduce the amount of memory required there can be a single disjunctive node for each predicate in the unnormalized program, taking input from the node for each rule concluding the predicate.

9 Future Work

It is possible to redefine the alternating fixpoint to reduce the number of tokens required for underestimates by starting each underestimate with the results of the previous one, as shown below.

$$
\begin{array}{lcll}
O_{0,\infty} & = & H & \\
O_{x,0} & = & E & (x > 0) \\
O_{x,y} & = & O_{x,y-1} \cup I(O_{x,y-1}, U_{x-1,\infty}) & (x > 0 \text{ and } y > 0) \\
U_{0,0} & = & E & \\
U_{x,0} & = & U_{x-1,\infty} & (x > 0) \\
U_{x,y} & = & U_{x,y-1} \cup I(U_{x,y-1}, O_{x,\infty}) & (y > 0)
\end{array}
$$

Using this definition, the area of coverage for an underestimate token $u(p, x, y)$ includes every $(x, y1)$ for $y1 > y$ and every $(x1, y1)$ for every $x1 > x$ and every $y1$, including $y1 < y$. This means that only one underestimate token per atom is required, whereas the previous method generated a new one whenever the shrinking overestimates allowed a shorter proof.

We have not found a similar optimization for the overestimates, but have not found a convincing reason why it would not be possible.

Another interesting area of investigation is the use of magic sets [BMSU85, BR87] with incremental updates. This would allow for incremental updates to the magic seed relations as well as incremental updates to the EDB, and could be useful within the scope of a transaction to continually verify constraints.

General logic programs with function symbols pose problems for bottom-up evaluation because they involve predicates with infinite extensions. AFP works for general logic programs and will terminate if no predicate has an infinite extension. When it terminates, the answers produced are correct. We would like to explore the implementation issues in supporting general logic programs.

10 Summary

AFP is the first incremental algorithm to correctly handle retraction, with the elimination of self-supporting conclusions, and to correctly handle negation, as defined by the well-founded semantics. We believe that AFP is well suited to data-driven applications where incremental recomputation of results is important.

The key advance in AFP is the data representation, which allows a computation to proceed monotonically forward without getting stuck in recursive loops, and which has a finite representation for atoms in alterate inner fixpoints. We have shown that this data representation, along with the AFP algorithm, is correct and demonstrated its feasibility with an implementation in Prolog.

11 References

[BMSU85] Francois Bancilhon, David Maier, Yehoshue Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 4th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1985.

[BR87] Beeri and Ramakrishnan. On the power of magic. In *Proceedings of the 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1987.

[BR88] Fancois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In Jack Minker, editor, *Deductive Databases and Logic Programming*, pages 439–517. Morgan Kaufmann, 1988.

[Cla78] K. L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[For79] Charles L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, CMU, 1979.

[For81] Charles L. Forgy. OPS5 reference manual. Technical Report CMU-CS-81-135, CMU, 1981.

[Mir87] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of the National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, August 1987.

[MJF+90] Daniel P. Miranker, Bernie J. Lofaso Jr., Gary Farmer, Arun Chandra, and David Brant. On a TREAT-based production system compiler. In *Proceedings of the Tenth International Conference on Expert Systems and Their Applications*, 1990.

[NGR88] Nayak, Gupta, and Rosenbloom. Comparison of RETE and TREAT production systems for Soar. In *Proceedings of the National Conference on Artificial Intelligence*, page 693, 1988.

[PP90] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Approaches to Artificial Intelligence: A Sourcebook.* North-Holland, New York, 1990.

[Rei78] R. Reiter. On closed world databases. In Gallaire and Minker, editors, *Logic and Databases.* Plenum Press, New York, 1978.

[Ros89] K. A. Ross. A procedural semantics for well-founded negation in logic programs. In *Eighth ACM Symposium on Principles of Database Systems*, pages 22–33, 1989.

[Ros90] Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings of the 9th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1990.

[Ull89] Jeffrey D. Ullman. Bottom-up beats top-down for DATALOG. In *Eighth ACM Symposium on Principles of Database Systems*, 1989.

[VG89] Allen Van Gelder. The alternating fixed point of logic programs with negation. Technical Report UCSC-CRL-89-39, UCSC, 1989.

[VGRS88] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics of logic programs. Technical Report UCSC-CRL-88-16, UCSC, 1988.

[VGT89] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus queries. Technical Report UCSC-CRL-89-40, UCSC, 1989.