An Operational Semantics for Network Datalog

Vivek Nigam[1], Limin Jia[2], Anduo Wang[1], Boon Thau Loo[1], Andre Scedrov[1]

[1] University of Pennsylvania, Philadelphia,USA
and [2] Carnegie Mellon University, Pittsburgh, USA

Abstract

Network Datalog (*NDlog*) is a recursive query language that extends Datalog by allowing programs to be distributed in a network. In our initial efforts to formally specify *NDlog*'s operational semantics, we have found several problems with the current evaluation algorithm being used, including unsound results, unintended multiple derivations of the same table entry, and divergence. In this paper, we make a first step towards correcting these problems by formally specifying a new operational semantics for *NDlog* and proving its correctness for the fragment of non-recursive programs. We also argue that if termination is guaranteed, then the results also extend to recursive programs. Finally, we identify a number of potential implementation improvements to *NDlog*.

1 Introduction

Declarative networking [10, 11, 12, 13] is based on the observation that network protocols deal at their core with using basic information locally available, *e.g.*, neighbor tables, to compute and maintain distributed states, *e.g.*, routes. In this framework, network protocols are specified using a declarative logic-based recursive query language called *Network Datalog* (*NDlog*), which can be seen as a distributed variant of Datalog [21]. In prior work, it has been shown that traditional routing protocols can be specified in a few lines of declarative code [13], and complex protocols such as Chord distributed hash table [23] in orders of magnitude less code [12] compared to traditional imperative implementations. This compact and high-level specifications enable rapid prototype development, ease of customization, optimizability, and the potentiality for protocol verification. When executed, *NDlog* programs result in efficient implementations, as demonstrated in open-source implementations [20, 22].

An inherent feature in networking is the change of local states due to usually small and incremental changes in the network topology. For example, a node might need to change its local routing tables whenever a preferred connection becomes available or when it is no longer available. Reconstructing a node's local state from scratch whenever there is a change in topology is impractical, as it would incur unnecessarily high communication overhead. For instance, in the path-vector protocol used in Internet routing, recomputation from-scratch would require all nodes to exchange all routing information, including those that have been previously propagated.

Therefore in declarative networking, nodes maintain their local states incrementally as new route messages are received from their neighbors. In literature, there are well known techniques for maintaining databases incrementally [8], in the form of *materialized views*, based on the traditional *semi-naïve* (SN) [3] evaluation strategy for Datalog programs. In order to accommodate these techniques to a distributed setting, Loo *et al.* in [10] proposed a *pipelined semi-naïve* (PSN) evaluation strategy for *NDlog* programs. PSN relaxes SN by allowing a node to change its local state by following a local pipeline of update messages. These messages specify the insertions and deletions scheduled to be performed to the node's local state.

Due to the complexity of combining incremental database view maintenance with data and rule distribution, until now, there is no formal specification of PSN nor a correctness proof. As PSN allows each node to compute its local fixed point and disregard global update ordering,

PSN does not necessarily preserve the semantics of the centralized SN algorithm. However, in a distributed setting, centralized SN evaluation is not practical. Therefore, studying the correctness properties of a distributed SN evaluation is crucial to the correctness of declarative networking.

In this paper, we aim to give formal treatment of the operational semantics of PSN and to prove its correctness. In the process, we identify several problems with PSN, namely, that it can yield unsound results; it can diverge; and it can compute the same derivation multiple times. In order to address these deficiencies, we present a new evaluation algorithm for *NDlog* called PSN^{ν} and prove its correctness for the fragment of non-recursive programs. We formalize both PSN^{ν} and SN algorithms computations by using the same set of transition rules. Then, we show that a PSN^{ν} execution for a distributed *NDlog* program derives the same facts as an SN execution for the same Datalog program in a centralized setting. This property is proved by showing that a PSN^{ν} computation run can be transformed into an SN computation run and vice-versa without affecting the resulting state. We also argue that the same reasoning is applicable to proving correctness of PSN^{ν} for recursive programs provided that PSN^{ν} terminates in the presence of messages inserting and deleting the same tuple. Finally, we identify several potential implementation improvements by using PSN^{ν}.

The rest of the paper is organized as follows. In Section 2, we review the basics of *NDlog* and of a simple SN algorithm used to maintain states incrementally in a centralized setting. Then, in Section 3 we review the PSN algorithm, explain the problems of PSN, and informally introduce PSN^{ν}. In Section 4, we formalize the operational semantics of PSN^{ν} in the form of a transition system and prove its correctness with respect to the SN algorithm. We have also formalized the operational semantics of *NDlog* in linear logic with subexponentials [19]. However, due to space constraints, we omit the formalisms from this paper, but details can be found in the extended version of this paper [18]. The linear logic encoding has many advantages over the state transition system presented here, which we discuss in Section 5. Finally in Section 6, we comment on related work and conclude by pointing to future work in Section 7.

2 Preliminaries

In this section, we review the language *Network Datalog* (*NDlog*) [10], which extends Datalog programs by allowing one to distribute Datalog rules in a network. Moreover, we also review an algorithm that maintains views incrementally in a centralized setting. This algorithm based on the semi naïve evaluation strategy will be used as the basis for showing correctness of the distributed algorithm that we propose later in Section 3.

2.1 Background: Datalog

We first review some standard definitions of Datalog, following [21]. A *Datalog* program consists of a (finite) set of logic rules and a query. A rule has the form $h(\vec{t})$ `:-` $b_1(\vec{t_1}), \ldots, b_n(\vec{t_n})$, where the commas are interpreted as conjunctions and the symbol `:-` as reverse implication; $h(\vec{t})$ is an atom called the head of the rule; $b_1(\vec{t_1}), \ldots, b_n(\vec{t_n})$ is a sequence of atoms and function relations called the body; and the $\vec{t}$s are vectors of variables and ground terms. Any free variable in a Datalog rule are is assumed to be universal quantified. *Function relations* are simple operations such as boolean, or arithmetic (*e.g.*, $X_1 < X_2$), or list manipulations operations (*e.g.*, `f_concat(S,P2)`). Semantically the order of the elements in the body does not matter, but it does have an impact on how programs are evaluated (usually from left to right). The query is a ground atom. We say that a predicate p depends on q if there is a rule where p appears in its head and q in its body.

The *dependency graph* of a program is the transitive closure of the dependency relation using its rules. We say that a program is (*non*)*recursive* if there are (no) cycles in its dependency graph. As a technical convenience, we assume that if predicates have different arities, then they have different names[1]. We classify the predicates that do not depend on any predicates as base predicates, and the remaining predicates as derived predicates. Consider the following non-recursive Datalog program where `p`, `s`, and `t` are derived predicates and `u`, `q`, and `r` are base predicates: {`p :- s, t, r; s :- q; t :- u; q :-; u :-`}. The set of all the ground atoms that are derivable from this program, called *view* or state, is the multiset {`s, t, q, u`}. In this particular example, each predicate is supported by only one derivation. If we added, however, the clause `s :- u` to this program, then the resulting view program would change to {`s, s, t, q, u`} where `s` appears twice, as there are two different ways to derive it: one by using `q` and another by using `u`.

Datalog's predicates (atoms) correspond to tuples in databases, and logical conjunction is equivalent to a join operation in database. For the rest of the paper, these terms are used interchangeably.

2.2 Network Datalog by Example

To illustrate *NDlog* programs, we provide an example based on a simplified version of the *path-vector* protocol, a standard routing protocol used for computing paths between any two nodes in the network. This protocol is used as a basis for Internet routing today, where different *autonomous systems* (or *Internet Service Providers*) exchange routes using this protocol.

```
r1 path(@S,D,P,C) :- link(@S,D,C), P=f_init(S,D).
r2 path(@S,D,P,C) :- link(@S,Z,C1), path(@Z,D,P2,C2), C=C1+C2, P=f_concat(S,P2), f_inPath(P2,S)=false.
```

The program takes as input `link(@S,D,C)` tuples, where each tuple represents an edge from the node itself (`S`) to one of its neighbors (`D`) of cost `C`. *NDlog* supports a *location specifier* in each predicate, expressed with "`@`" symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, `link` tuples are stored based on the value of the `S` attribute.

Rules `r1-r2` recursively derive `path(@S,D,P,C)` tuples, where each tuple represents the fact that there is a path `P` from `S` to `D` with cost `C`. Rule `r1` computes one-hop reachability, given the neighbor set of `S` stored in `link(@S,D,C)`. Rule `r2` computes transitive reachability as follows: if there exists a link from `S` to `Z` with cost `C1`, and `Z` knows a path `P2` to `D` with cost `C2`, then `S` can reach `D` via the path `f_concatPath(S,P2)` with cost `C1+C2`. Rules `r1-r2` utilize two list manipulation functions: `P= f_init(S,D)` initializes a path vector with two nodes `S` and `D`, while `f_concatPath(S,P2)` prepends `S` to path vector `P2`. To prevent computing paths with cycles, rule `r2` uses the function `f_inPath`, which returns true if `S` is in the path vector `P`. Notice that although this function seems to contain a negation, it is in fact implemented without using it: if we assume nodes to be specified using natural numbers, then one can write a specification for `f_inPath` that does not use negation, but that uses the inequalities $<$ and $>$.

To implement the path-vector protocol in the network, each node runs the exact same copy of the above program, but only stores tuples relevant to its own state. What is interesting about this program is that predicates in the body of rule `r2` have different location specifiers indicating that they are stored on a different node. To improve performance and eliminate unnecessary communication, we use a *rule localization* [10] rewrite procedure that transforms a program into an equivalent one where all elements in the body of a rule have the same location, but the head of the rule may reside at a different location than the body predicates. We call a

[1] One can easily rewrite predicate names and distinguish them by using their arities.

rule non-local when two body atoms have different location specifiers. For instance, the clause `r1` above is local, while the clause `r2` is not. The rule localization procedure rewrites the clause `r2` to the following two clauses that are local:

```
r2-1: path(@S,D,P,C) :- link(@S,Z,C1), aux(@S,Z,D,P2,C2), C=C1+C2, P=f_concat(S,P2), f_inPath(P2,S)=false.
r2-2: aux(@S,Z,D,P,C) :- path(@Z,D,P,C), link(@Z, S).
```

Here, the predicate `aux` is a new predicate, that is, it does not appear in the original alphabet of predicates. As specified in the rule `r2-1`, this predicate is used to inform all neighbors, `S`, of the node `Z` of the existence of a path `P` with cost `C` from a node `Z` to a node `D`. It is not hard to show, by induction on the height of derivations, that this program is equivalent to the previous one in the sense that a `path` tuple is derivable using one program if and only if it is derivable using the other. For the rest of this paper, we assume that such localization rewrite has been performed.

2.3 Maintaining Views Incrementally

Given a datalog program and a set of base tuples or facts, one derives all possible facts that can be derived from the logic program by using bottom-up evaluation algorithms [21]. For instance, the view of the path example above would consist of all possible paths in the network. However, consider now that there is a change on the set of base facts, for instance, when a new link in the network has been established or an old link has been broken. In this case, one would need to update the view of the database in order to accommodate the changes in the base predicates. One way of doing so is to forget all derived tuples and rederive the new view from scratch. Since the changes to base predicates do not necessarily affect the derivations of all facts in the original view of the database, starting from scrath might involve repeating unecessarily the same work. A better way is to maintain a view *incrementally*, where one only takes into account the facts that are affected by the changes to the base predicates, while the rest of the facts remain untouched.

Algorithm 1 is such an algorithm based on the traditional Semi-naïve (SN) evaluation strategy that maintains a database incrementally when given a set of changes to base predicates [8]. Semi-naïve (SN) evaluation iteratively updates the view until a fixed point is reached. Tuples computed for the first time in the previous iteration are used as input in the current iteration; and new tuples that are generated for the first time in the current iteration are then used as input to the next iteration.

First, we create for each rule $h(\vec{t})$ `:-` $b_1(\vec{t_1}), \ldots, b_n(\vec{t_n})$ in a Datalog program the following delta insertion and deletion rules, where we use the names INS and DEL to denote an insertion and deletion, respectively:

$\texttt{INS}(h(\vec{t}))$ `:-` $b_1^\nu(\vec{t_1}), \ldots, b_{i-1}^\nu(\vec{t_{i-1}}), \Delta b_i(\vec{t_i}), b_{i+1}(\vec{t_{i+1}}), \ldots, b_n(\vec{t_n})$

$\texttt{DEL}(h(\vec{t}))$ `:-` $b_1^\nu(\vec{t_1}), \ldots, b_{i-1}^\nu(\vec{t_{i-1}}), \Delta b_i(\vec{t_i}), b_{i+1}(\vec{t_{i+1}}), \ldots, b_n(\vec{t_n})$

We start initially with two copies of the view, one marked with ν, corresponding to the predicates with the ν superscript, and another not marked with ν. Then, given a set of insertions, I_k, and deletions, D_k, for each base predicate, p_k, Algorithm 1 uses the delta-rules above to incrementally maintain the view. Intuitively, an update of a predicate, b_i, is removed from one of these sets and it is stored in its corresponding delta predicate, Δb_i. Then, any update derived by any delta-rule in one iteration is processed in the following iteration until no more updates are derived. The operations *insert* and *remove*, respectively, inserts and removes (non-deterministically) an element from a set, while the operation *flush* removes all elements from one set and inserts them into another set. The algorithm proceeds as follows: If we are in, say, the $i^{th}+1$ iteration, then the contents of the table without ν corresponds to the view at the $i^{th}-1$ iteration and the contents of the table with ν to the view at the i^{th} iteration. The

$i^{th}+1$ iteration consists of executing the delta-rules for all updates in I_k and D_k: First, one picks (non-deterministically) an element from either the set of insetions, I_k, or of deletions, D_k, and uses accordingly the set of insertion or deletion delta rules. Then, whenever an insertion or deletion rule is fired, we store the derived tuple in I_k^ν and D_k^ν respectively. Finally, once all rules have been executed, we change the view accordingly and proceed to the next iteration, but now using the updates stored in I_k^ν and D_k^ν, which correspond to the updates derived in iteration $i^{th}+1$. This is done by the instructions in the for loop which use *set-operations*.

Algorithm 1 SN-algorithm.

while $\exists I_k.size > 0$ or $\exists D_k.size > 0$ **do**
  while $\exists I_k.size > 0$ or $\exists D_k.size > 0$ **do**
    $\Delta t_k \leftarrow I_k$.remove (resp. $\Delta t_k \leftarrow D_k$.remove)
    $I_k^{aux}.insert(\Delta t_k)$ (resp. $D_k^{aux}.insert(\Delta t_k)$)
    execute all insertions (resp. deletion) delta-rules for t_k:
      $\Delta p_k^{i+1} \leftarrow p_1^\nu, \ldots, p_{i-1}^\nu, \Delta t_k, p_{k+1}, \ldots, p_n$
    for all derived tuples $p \in \Delta p_k^{i+1}$ **do**
      $I_k^\nu.insert(p)$ (resp. $D_k^\nu.insert(p)$)
    end for
  end while
  for all predicates p_j **do**
    $p_j \leftarrow (p_j \cup I_j^{aux}) \setminus D_j^{aux}$; $p_j^\nu \leftarrow (p_j \cup I_j^\nu) \setminus D_j^\nu$; $I_j \leftarrow I_j^\nu.flush$; $D_j \leftarrow D_j^\nu.flush$;
    $D_j^{aux} \leftarrow \emptyset$; $I_j^{aux} \leftarrow \emptyset$; $\Delta p_j^{i+1} \leftarrow \emptyset$
  end for
end while

Algorithm 1 maintains correctly the view of a Datalog program [8] whenever there is exactly one derivation for any tuple. This limitation is due to the use of set semantics. Despite of this limitation, Algorithm 1 captures most of the programs used until now in declarative networking. For instance, we can use it to maintain the datalog program for path vector program described above since any `path` tuple is supported by just one derivation. There are other more complicated algorithms that can also maintain views of programs where tuples have multiple supporting derivations. However, formalizing these algorithms seems to be a non-trivial task and is left for future work.

3 Network Datalog Program Execution

Maintaining views incrementally in a distributed setting, however, generates many challenges. While in the centralized setting one can enforce a high degree of synchronization, in a distributed setting this is in general not the case. For example, in Algorithm 1 one processes older updates always before newer ones. On the other hand, in a distributed setting an agent is not usually required to stop processing a newer update until all other agents in the system have processed older updates. Such synchronization would make the system unfeasible in practice [10].

Guaranteeing desirable properties, such as termination, in such an asynchronous setting is usually much harder than in centralized setting. We review in this section the distributed evaluation algorithm currently used by *NDlog* called *pipelined semi-naïve* (PSN). We identify some problems with this algorithm and then propose a new evaluation algorithm called PSN^ν.

3.1 Problems in Pipelined Semi-naïve Evaluation

In order to maintain incrementally the states of nodes or agents in a distributed setting and at the same time avoid synchronization among them, Loo *et al.* in [10, 11] proposed PSN. In

```
@1: {}[]                    {p}[INS(p)]                  {p}[INS(p)]              {p}[]
@2: {r,s,t}[INS(r)]         {r,s,t}[]                    {r}[DEL(s),DEL(t)]       {r}[]
@3: {}[DEL(q)]   -- INS(r)--> {}[DEL(q)] -- DEL(q),DEL(u)--> {}[]              ---->* {}[]
@4: {}[DEL(u)]              {}[DEL(u)]                   {}[]                     {}[]
```

Figure 1: PSN computation-run resulting in an incorrect final state. The i^{th} row depicts the evolution of the view, in curly-brackets, and the queue, in brackets, of node i. The updates in the arrows are the ones dequeued by PSN and used to update the view of the nodes. We also elide the `@` in the predicates and updates.

PSN, each agent has a queue of *messages* scheduling insertions and deletions of tuples to the agents's local state. An agent proceeds in a similar fashion as in Algorithm 1; it dequeues one update; then executes its corresponding insertion or deletion delta-rules; and then for each derived tuple, it sends a message which is to be stored at the end of the queue of the node specified by derived tuple's location specifier (`@`).

However, when a message reaches a node, it is not only stored at the end of the node's queue, but also immediately used to update the node's local state, that is, the tuple in the message is immediately inserted into or deleted from the node's view. We now demonstrate that updating a node's view by using messages before they are dequeued can yield unsound results. Consider the following *NDlog* program, which is the same program shown in Section 2.1, but now distributed over four nodes. The view of this program is {s@2, t@2, q@3,u@4}:

```
p@1 :- s@2, t@2, r@2     s@2 :- q@3     t@2 :- u@4     q@3 :-     u@4 :-
```

Consider as well the PSN computation-run depicted in Figure 1, which uses the messages inserting the tuple `r@2` and deleting the tuples `q@3` and `u@4`. Notice that in the first state these updates have already been used to update the view of the nodes as described in PSN. In the final transitions, none of the updates deleting `s` or `t` trigger the deletion of `p` because the bodies of the respective deletion rules are not satisfied since `t` and `u` are no longer in node 2's view. Hence, the predicate `p` is entailed after PSN terminates although it is not supported by any derivation.

The second problem that we identify is that unlike SN, PSN does not avoid redundant computations. This is because in PSN a delta-rule is fired by using the contents currently stored in a node's view, and not distinguishing, as in SN, its two previous states, which in SN is accomplished by using the predicates p and p^{ν}. For example, the *NDlog* rule `p@1 :- t@1, t@1` would be rewritten into the following two insertion rules, where we elide the `@` symbols: `INS(p) :-` Δ`t, t` and `INS(p) :- t,` Δ`t`. Thus if we dequeue an update inserting the tuple `t`, both rules are fired, and two instances inserting `p` are added to the queue of node 1.

Finally, the third problem that we identify is divergence. Consider the simple *NDlog* program composed of two rules: `p@1 :- a@1` and `p@1 :- p@1`; and that the queue of node `1` is `[INS(a),DEL(a)]`. The insertion (resp. deletion) of `a` will cause an insertion (resp. deletion) of `p` to be added at the end of the queue. Because of the second rule, the insertion and deletion of `p` will propagate indefinitely many insertions and deletions of `p` and therefore causing PSN to diverge.

In the informal description of PSN, presented in [10, 11], many assumptions were used, such as that messages are not lost; a *Bursty Model*, that is, the network eventually *quiesces* (does not change) for a time long enough for all the system to reach a fixed point; that message channels are FIFO, hence no reordering of messages is allowed; and that timestamps are attached to tuples in order to evaluate delta rules. Even under these strong assumptions, the problems in PSN mentioned above persist. What is more troublesome is that this design is reflected in the current implementation of *NDlog* and therefore, all *NDlog* programs exhibit those flaws.

In the next section, we propose a new evaluation algorithm, called *PSN*$^{\nu}$, which not only

corrects these problems, but also does not require the last two assumptions (FIFO channels and use of timestamps). The removal of these two assumptions not only simplifies the implementation, but also potentially leads to improved performance, since the implementation no longer requires receiver-based network buffers to guarantee in-order delivery of messages.

3.2 New Pipelined Semi-naïve Evaluation

At a high-level, PSN^{ν} works as follows: Instead of using queues to store unprocessed updates, we use a single *bag*, denoted as $\mathcal{U}$, that specifies the asynchronous behavior in the distributed setting by abstracting the order in which updates are used. Thus in this abstraction, we do not need to take into account the @ specifiers since all messages go to $\mathcal{U}$. One processes *NDlog* rules into delta-rules exactly as in the SN algorithm, so that the multiple derivation problem does not occur. Then, one PSN^{ν}-iteration is completed by executing in a sequence the following three basic commands, which preserve the invariant that before and after a PSN^{ν}-iteration the views of the tables with ν and without ν are the same:
pick – One picks (non-deterministically) any update, u, from the bag $\mathcal{U}$, except if u is a deletion of an atom that is not (yet) in the view. Then, if u is an insertion of predicate p, one inserts the corresponding p^{ν} to the ν table, otherwise if it is a deletion of the same predicate, one deletes p^{ν} from the ν table;
fire – After picking an update, one executes all the delta-rules corresponding to u. If a rule is fired, then one inserts the derived tuple into the bag $\mathcal{U}$.
update – Once all delta-rules are executed, one updates the view according to u: if u is an insertion or deletion of predicate p, one inserts it into or delete it from the view without ν.

The execution of an SN-iteration can also be specified with the use of the same three basic commands above. However, instead of applying just one sequence of the three commands, the $i^{th}+1$ SN-iteration is composed of three phases: first, all elements in $\mathcal{U}$ are picked using the *pick* command. The resulting contents in the ν table is updated with the updates derived in the previous iteration. Hence, the contents of the ν table correspond exactly to the view at the i^{th} iteration, while the contents in table without ν corresponds exactly to the view at the $i^{th}-1$ iteration, as in Algorithm 1. Then one executes the delta-rules for all updates picked in the previous phase, deriving and storing new updates in the bag $\mathcal{U}$. After this phase, $\mathcal{U}$ contains the updates derived at the $i^{th}+1$ iteration. Finally, in the third phase, one executes eagerly the *update* command which then updates the contents in table without ν to match the contents of the table with ν.

4 Transition rules for the basic commands

As previously discussed, differently from a centralized setting where one is allowed to enforce a great deal of synchronization on how updates are processed, in a distributed setting, it is no longer possible to do so. As a consequence, in a distributed setting, traditional ways to define semantics of logic programs, such as by using fixed point operators, *e.g.*, T_p operators, do not closely correspond to the operational semantics of bottom-up evaluations. For instance, SN is closely related to the T_p operator since at the end of each iteration the set of derived facts by these methods coincide. In a distributed setting, however, this is not the case since newer updates can be processed before older updates. This paper, therefore, takes a different approach and formalizes the operational semantics by using, instead, a transition system that can explain both the centralized and the distributed algorithms. Then we show that any trace in one can

be transformed into the other, showing hence that the distributed algorithm is correct with respect to the centralized one.

In order to formalize the operational semantics of the basic commands, we introduce the following basic definitions. An *update* is a tuple $\langle u, p\rangle$, where u is either INS or DEL denoting, respectively, an insert or a delete update, and p is the fact that is being inserted or deleted from the database.

Definition 1. *A state of the system is a tuple of the form $\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle$, where $\mathcal{K}$ is a multiset of facts, and $\mathcal{U}, \mathcal{P}$, and $\mathcal{E}$ are all multisets of updates.*

Intuitively, the multiset $\mathcal{K}$ contains both the view with and without ν, the multiset $\mathcal{U}$ is the bag of updates which have to be used to update an agent's view, the multiset $\mathcal{P}$ contains the updates that have been picked and are scheduled to be used to fire delta rules, and finally the multiset $\mathcal{E}$ contains the updates that have been already used to fire delta rules, but not yet used to update the view.

$$\frac{\langle \text{INS}, p(\vec{t})\rangle \in \mathcal{U}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \cup \{p^{\nu}(\vec{t})\}, \mathcal{U} \setminus \{\langle \text{INS}, p(\vec{t})\rangle\}, \mathcal{P} \cup \{\langle \text{INS}, p(\vec{t})\rangle\}, \mathcal{E}\rangle}\ \textit{pick}_I$$

$$\frac{\langle \text{DEL}, p(\vec{t})\rangle \in \mathcal{U} \quad \text{and} \quad p^{\nu}(\vec{t}) \in \mathcal{K}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{p^{\nu}(\vec{t})\}, \mathcal{U} \setminus \{\langle \text{DEL}, p(\vec{t})\rangle\}, \mathcal{P} \cup \{\langle \text{DEL}, p(\vec{t})\rangle\}, \mathcal{E}\rangle}\ \textit{pick}_D$$

$$\frac{u \in \mathcal{P} \quad \text{and} \quad \mathcal{F} = \textit{firRules}(u, \mathcal{K}, \mathcal{R})}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K}, \mathcal{U} \cup \mathcal{F}, \mathcal{P} \setminus \{u\}, \mathcal{E} \cup \{u\}\rangle}\ \textit{fire}$$

$$\frac{\langle \text{INS}, p(\vec{t})\rangle \in \mathcal{E}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \cup \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{INS}, p(\vec{t})\rangle\}\rangle}\ \textit{update}_I$$

$$\frac{\langle \text{DEL}, p(\vec{t})\rangle \in \mathcal{E}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{DEL}, p(\vec{t})\rangle\}\rangle}\ \textit{update}_D$$

Figure 2: Transition rules specifying the operational semantics for the basic commands. Here, we only use multiset operations; $\mathcal{R}$ is the set of rules in the program, and $\textit{firRules}(u, \mathcal{K}, \mathcal{R})$ is a function that returns the set of all updates, $\mathcal{F}$, that can be fired using the update u, the view $\mathcal{K}$, and the set of delta rules obtained from $\mathcal{R}$.

The transition rules specifying the operational semantics for the basic commands are depicted in Figure 2. The first two rules, $\textit{pick}_I$ and $\textit{pick}_D$, specify the *pick* command. One moves, respectively, an insertion and an deletion update from the bag of updates $\mathcal{U}$ to the bag of picked updates $\mathcal{P}$, and depending on the type of the update, a fact, $p^{\nu}(\vec{t})$, is either inserted into or deleted from the view. The third rule, *fire*, specifies the command *fire*, where we make use of the function *firRules*. Intuitively, this function takes an update, $\langle u, p(\vec{t})\rangle$, the current view, $\mathcal{K}$, and the set of rules, $\mathcal{R}$, as input and returns the set of all updates, $\mathcal{F}$, obtained from firing delta-rules that contain Δp in its body. This set is then added to the set $\mathcal{U}$ of updates that have to be processed later. Finally, the last two rules, $\textit{update}_I$ and $\textit{update}_D$, specify the operation of updating the view. Similar to the rules for *pick*, they either insert into or delete from the view a fact $p(\vec{t})$.

Notice that, in the rules for *pick*, one is not allowed to pick a delete update of a tuple that is not (yet) in the view of the database. This restriction is imposed in order to maintain the count of tuples consistent. Intuitively, a delete update is generated to delete a tuple that is

present in the view. However, due to the asynchronous behavior of the system, it can be the case that this tuple is not yet present in the view because the insert update that was going to do so has not yet been processed. Therefore, an agent needs to wait until the view contains the element being deleted before processing a delete update.

A *computation run* using a program $\mathcal{R}$ is simply a valid sequence of applications of these transition rules. We call the first state of a computation run the initial state and its last state the resulting state. Clearly, when all three multisets $\mathcal{U}, \mathcal{P}$, and $\mathcal{E}$ are empty, computation can no longer proceed (until a new change in the environment triggers new updates to base predicates). In this case, we say that all updates have been processed and the final view is obtained. In particular, we are interested in the computation runs that correspond to execution runs of SN and of PSN^{ν}, defined below.

Definition 2. *A computation run is a* complete execution *if it can be partitioned into a sequence of transitions of both* pick_I *and* pick_D*, followed by a sequence of transitions of* fire*, and finally a sequence of transitions of* update*, such that the multiset of tuples,* $\mathcal{T}$*, used by the sequence of* pick_I *and* pick_D *transitions is the same being used by the sequence of* fire *and* update *transitions. A complete iteration is an* SN-iteration *if* $\mathcal{T}$ *contains all updates at the initial state that are in* $\mathcal{U}$*. A complete iteration is a* PSN^{ν}-iteration *if* $\mathcal{T}$ *contains only one update.*

Definition 3. *We call a computation run a* PSN^{ν}*-execution (respectively SN-execution) if it can be partitioned into a sequence of* PSN^{ν}*-iterations (respectively SN-iterations) and where in the last state all updates have been processed.*

A computation run corresponding to an iteration of SN first picks all updates that appear in the initial state in $\mathcal{U}$, then deduces from them new updates by using the *fire* command, and updates the view using the same updates by using the *update* command. The new updates that are generated remain untouched until all old updates have been processed. Such iteration corresponds exactly to the inner while loop in Algorithm 1. On the other hand, a computation run corresponding to an iteration of PSN^{ν} picks only one update at a time and executes all three commands before picking another update.

In Figure 3, a PSN^{ν}-execution is depicted that uses the same initial condition as in the computation run depicted in Figure 1. Unlike the previous PSN computation, here, the final state is consistent since the resulting state contains only the fact `r` and not `p`. In order to relate this execution to the distributed setting, we just need to attach the location specifier `@` to the facts appearing in the set of updates, *e.g.*, `INS(p@1)`, and in the view, *e.g.*, `r@2`. So, in the final state the views of all agents are empty except the view of agent 2, which has `r@2`.

Although we use a single bag to store updates, in reality, agents in *NDlog* have their own buffers containing incoming updates. The use of a single bag, however, is enough for the purpose of showing that, despite of its asynchronous behavior, PSN^{ν} is sound and complete with respect to SN. Updates are picked non-deterministically from the bag of updates, $\mathcal{U}$, and in PSN^{ν}-executions, only one update is processed at a time. Therefore in PSN^{ν}-executions, one processes in an atomic step any update even if the update chosen is not the oldest one in the bag of updates. This contrasts with SN-executions where one is required to process all old updates before new ones are picked. We could easily extend our model, however, to accommodate the location of data by using a different bag for each agent and attach location specifiers to programs, updates, and facts. However, this would make the correctness proof unecessarily more complicated.

The following theorem establishes that PSN^{ν} is sound and complete with respect to SN.

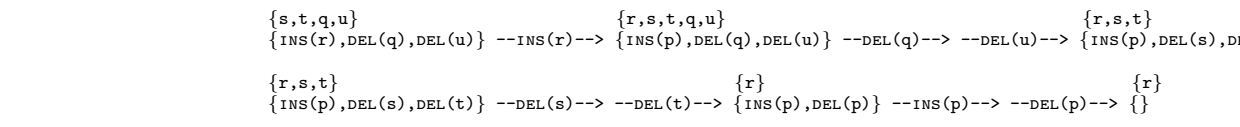

Figure 3: A PSN^{ν}-execution run. Here we elide some of the intermediary states.

Theorem 4. *Let $\mathcal{R}$ be a non-recursive Datalog program, and let $\mathcal{S}$ be an initial state. Then there is a PSN^{ν}-execution from $\mathcal{S}$ to a final state $\mathcal{S}'$ using $\mathcal{R}$ if and only if there is an SN-execution from $\mathcal{S}$ to the same final state $\mathcal{S}'$ also using $\mathcal{R}$.*

Proof (Sketch) We prove the theorem above by showing that: 1) we can permute two PSN^{ν}-iterations; 2) we can merge a complete-iteration and a PSN^{ν}-iteration into a larger complete-iteration; and 3) conversely we can split a larger complete-iteration into a smaller complete-iteration and a PSN^{ν}-iteration. Given a PSN^{ν}-execution, we construct an SN-execution by induction as follows: we use the first operation to permute downwards the PSN^{ν}-iteration that picks any element in the initial state's $\mathcal{U}$ set, then repeat it with its subexecution. The resulting execution has all PSN^{ν}-iterations in the same order as in an SN-execution. We merge them into SN-iterations by applying the second operation repeatedly. For the converse direction, given an SN-execution, we apply repeatedly the third operation to split SN-iterations and obtain a PSN^{ν}-execution. The complete proof appears in the extended version of this paper [18]. □

Corollary 5. *For non-recursive programs, a query is entailed by using PSN^{ν} iff it is entailed by using SN.*

In the proof of Theorem 4, we perform three different operations when transforming a PSN^{ν}-execution into an SN one. While performing these operations, however, it can happen that new rules are fired. In particular, this happens when we permute a PSN^{ν}-iteration that uses a deletion update over a PSN^{ν}-iteration that uses an insertion update. The updates generated in all cases are necessarily conflicting, that is, are pairs of insertions and deletions of the same tuple. In the general case, we cannot guarantee that PSN^{ν} terminates when processing such conflicting updates, but we can guarantee its termination if the program is non-recursive since these programs do not contain dependency cycles and therefore the propagation of updates will eventually end.

However, if we can guarantee such termination for PSN^{ν}, then the proof works exactly in the same way. For example, our path example, shown in Section 2, which is a recursive program, is such type of program. Because of the use of the function `f_inPath`, one does not compute paths that contain cycles. This restriction alone is enough to guarantee termination of PSN^{ν}: the number of `path`-updates propagated by conflicting updates inserting and deleting the same `link` tuple is finite. Therefore we can use the same reasoning above to show that PSN^{ν} is correct for this program.

Extending this work to larger classes is left for future work. We discuss more on this at the end of this paper.

5 Linear Logic with Subexponentials

Although in this paper we make use of a transition system to model *NDlog*'s operational semantics, mainly for presentation issues, we in fact have formalized all definitions above in terms of logic. More specifically, we encode the transition system used above in linear logic [7] extended with subexponentials [19]. The details of the encoding can be found in the technical report [18]. Since linear logic is a precise and well-established language, used already for both specifying and reasoning about semantics of programming languages, we expect to capitalize on the existing

work to improve the correctness result above to wider classes of *NDlog* programs. Moreover, linear logic also provides us with a finer detail on how data is manipulated, for instance, the function *firRules* is also specified in greater detail in our linear logic specification.

We have also observed that linear logic when extended with subexponentials seems to be a suitable framework for specifying systems where resources/data move from one place to another. We briefly discuss this matter.

In linear logic, due to the exponentials that control the availability of the structural rules, one distinguishes between two kinds of formulas: the linear ones and the unbounded ones. Linear formulas can be interpreted as resources that can only be used once, while the unbounded formulas can be interpreted as program instructions that can be used as many times as needed. Under this resource aware interpretation of linear logic, one can specify, for example, systems where agents manipulate the state of the world. A state is specified as a collection of linear formulas and an action of an agent is specified by a set of unbounded formulas. This distinction among formulas is reflected on syntax by using two different contexts one for the linear formulas and another for unbounded formulas. Thus in plain linear logic, there is no natural means to distinguish resources that are located in one place from another. All resources are placed in the same linear context.

However, the linear logic exponentials are not canonical [4]. In fact, we can assume a system with as many exponential-like operators, called subexponentials [19], as we need which may or may not allow weakening and/or contraction. Thus with subexponentials, instead of only two contexts as in linear logic, one can accommodate as many contexts as one needs. For instance, one can use a particular (linear) context to store resources, that is, linear formulas, that belong to some agent and in another context to store the resources available to another agent. This tight correspondence between distributed systems and linear logic with subexponentials seems to be interesting and it seems worth to further pursue more connections, in particular, involving verification techniques.

6 Related Work

Navarro *et al.* proposed in [17] an operational semantics for a variation of the *NDlog* language that also includes rules with events. However, their semantics also computes unsound results and therefore it is not suitable as an operational semantics for *NDlog*. For instance, besides the problems we identify for PSN, one is also allowed in their work to pick an update that deletes an element without checking if this element is present in the view, which may also yield unsound results. Moreover, in their operational semantics no incremental maintenance algorithm is incorporated. Therefore, users of their language are required to implement themselves how states are updated when incoming updates arrive at any node and furthermore prove its correctness.

Although here we focus on declarative networking, maintaining states incrementally in a distributed setting can also be useful when programming robots. As nodes in a network, robots are usually in an environment that changes incrementally, for example, objects move from one place to another. Since robots perform actions by taking into account the facts that they believe to be true at that moment, for the robot to perform sound actions, their internal knowledge bases have to be maintained correctly and efficiently whenever they detect changes in the environment. Ashley-Rollman *et al.* proposed a language designed for programming robots and inspired by *NDlog* called MELD [2]. Although the operational semantics of their language seems to agree with PSN^{ν}, we are not aware of any formal specification of its operational semantics nor of any correctness proof. We believe that their language will also benefit from the insights and results obtained here.

Adjiman *et al.* in [1] use classical propositional logic to specify knowledge bases of agents in a peer-to-peer setting. They prove correct a distributed algorithm that computes the consequences of inserting a literal, that is, an atom or its negation, to a node (or peer). Since they use resolution in their algorithm, they are able to deduce not only the atomic formulas that are derivable when an insertion is made, but propositional formulas in general. A fundamental difference from our appproach seems to be that while they are mainly interested in finding the consequences resulting from inserting a formula, here, we are interested in efficiently maintaining a set of consequences that was previously derived. In particular, after a set of consequences is derived using a first insertion, it is not clear in their approach how to update these consequence when a second insertion is made.

Linear logic has previously been used to specify concurrent systems [14, 15]. For instance, one is able to encode in linear logic many formalisms that are used to specify distributed sytems, for example the π-calculus, Petri-nets, Concurrent ML, and other distributed systems. Linear logic has also been used to specify access control policies [6]. One is able, for instance, to express policies that are not permanent but consumable, for example, a one-time access to a room. In a proof-authorization code framework, whenever a client, such as a mobile phone, requests a server for access to some resource, it attaches a linear logic proof demonstrating that his request follows from the given policies. In all of these approaches, however, it does not seem possible to encode located resources in a natural way as when using linear logic with subexponentials. In particular, it seems that in plain linear logic one always needs to rely on terms, such as lists or constants, to encode the notion of located resources. Here on the other hand, we encode located resources in the level of propositions by using subexponentials.

7 Conclusions and Future Work

In this paper, we have developed a new PSN algorithm, PSN^{ν}, which is key to specifying the operational semantics of *NDlog* programs. We have proven that PSN^{ν} is correct with regard to the centralized SN by showing that an SN execution can be transformed into a PSN^{ν} execution and vice versa. In the extended version of this paper, we have given a well-defined operational semantics for PSN^{ν} using linear logic with subexponentials. Furthermore, PSN^{ν} lifts restrictions such as FIFO channels from *NDlog* implementations and leads to potential performance improvements of protocol execution.

This work is part of a bigger effort to formally analyze network protocol implementations [5, 24]. The results in this paper lay a solid foundation toward closing the gap between verification and implementation. An important part of our future work is to formalize low-level *NDlog* implementations so that verification results on high-level specifications can be applied to low-level implementations.

In our correctness proof, we limited ourselves to the fragment of non-recursive programs. The main problem of including larger classes of programs is that we cannot necessarily guarantee termination of PSN^{ν} using recursive programs in the presence of conflicting updates, that is, updates inserting and deleting the same fact. However, if we can guarantee such termination for PSN^{ν}, then the proof works exactly in the same way. Moreover, since the SN algorithm that we use in this paper is only shown to be correct when tuples have at most one supporting derivation, the correctness of PSN^{ν} is also restricted to this fragment. Given these restrictions, we believe that there are many directions to extend the class of programs considered in this paper:

• Non-recursive programs where facts can have multiple supporting derivations: It seems possible to modify Algorithm 1 in such a way that the resulting algorithm also works correctly for

this class of programs. For instance, instead of using set semantics, one could attempt to use multiset semantics, where one not only keeps track of which facts have been deduced, but also of the number of derivations supporting it. Then extending PSN^{ν} to accommodate this change seems to be straighforward. The programming language MELD seems to go a step further in this direction and also store the depth of the derivations supporting a fact. This allows one to perform further optimizations of the operational semantics of their language, such as deciding whether or not to process an update according to the depth of the derivation supporting it.

• Recursive programs where tuples have a finite number of supporting derivations: As discussed before, we conjecture that if all facts have always a finite number of supporting derivations, then we can guarantee termination of PSN^{ν} whenever SN terminates and they would yield the same result. The proof would work exactly in the same way. Some work on the problem of determining when all facts derived from a Datalog program have a finitely many supporting derivations has appeared in literature [16]. It seems possible to adapt such approach to a distributed setting when using provenance mechanisms [9].

• Recursive programs where tuples can have infinitely many supporting derivations: It seems that in this case one cannot avoid divergence unless some level of synchronization among agents is allowed: Before processing an insert update, an agent would need to confirm with its neighbor agents that all previous updates have already been processed. If this is the case, then the agent checks its current bag of updates and cancels up any conflicting updates. Another idea is to use provenance mechanisms [9] as suggested in the previous case. Fortunately, however, until now no real applications required such type of programs.

Finally, we still need to investigate precisely how to handle aggregates and negation in a distributed setting. It seems to be possible to incorporate well-known techniques [8] that maintain states in the centralized setting into PSN^{ν}.

We plan to continue pursuing all of these directions in the near future.

Acknowledgments: Scedrov, Loo, Nigam, and Wang were partially supported by AFOSR MURI "Collaborative policies and assured information sharing". Additional support for Scedrov and Nigam from ONR Grant N00014-07-1-1039 and from NSF Grants CNS-0524059 and CNS-0830949. Wang was also partially supported by NSF CAREER CNS-CNS-0845552.

References

[1] Philippe Adjiman, Philippe Chatalic, François Goasdoué, Marie-Christine Rousset, and Laurent Simon. Distributed reasoning in a peer-to-peer setting: application to the semantic web. *J. Artif. Int. Res.*, 25(1):269–314, 2006.

[2] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *IROS*, pages 2794–2800. IEEE, 2007.

[3] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Prog, 4(3):259–262*, 1987.

[4] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713, pages 159–171. Springer, 1993.

[5] Formally Verifiable Networking. `http://netdb.cis.upenn.edu/fvn/`.

[6] Deepak Garg, Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. A linear logic of authorization and knowledge. In *ESORICS*, pages 297–312, 2006.

[7] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[8] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[9] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, pages 1108–1119. IEEE, 2009.

[10] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[11] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. In *Communications of the ACM (CACM)*, 2009.

[12] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[13] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[14] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 35–46. ACM, 2005.

[15] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, September 1996.

[16] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. In *Australian Database Conference*, pages 274–288, 1993.

[17] Juan A. Navarro and Andrey Rybalchenko. Operational semantics for declarative networking. In *PADL*, pages 76–90, 2009.

[18] Vivek Nigam, Limin Jia, Anduo Wang, Boon Thau Loo, and Andre Scedrov. An operational semantics for network datalog. Extended version available at `http://netdb.cis.upenn.edu/fvn/ndlogsemantics.pdf`, January 2010.

[19] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP*, pages 129–140, 2009.

[20] P2: Declarative Networking System. `http://p2.cs.berkeley.edu`.

[21] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[22] RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`.

[23] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[24] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally Verifiable Networking. In *SIGCOMM HotNets-VIII*, 2009.