An Operational Semantics for Network Datalog

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

[1] University of Pennsylvania, Philadelphia, USA
{vnigam,scedrov}@math.upenn.edu, {anduo,boonloo}@seas.upenn.edu
[2] Carnegie-Mellon University, Pittsburg, USA
liminjia@cmu.edu

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 used, including unsound results, unitended 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. Our formalization uses linear logic with subexponentials. 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 [7–10] 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 [16]. In prior work, it has been shown that traditional routing protocols can be specified in a few lines of declarative code [10], and complex protocols such as Chord distributed hash table [18] in orders of magnitude less code [9] 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, these declarative networks result in efficient implementations, as demonstrated in open-source implementations [15, 17].

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 [6], in the form of *materialized views*, based in the traditional *semi-naïve* (SN) [2] evaluation strategy for Datalog programs. In order to accommodate these techniques to a distributed setting, Loo *et al.* in [7] 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, specifying the insertions and deletions scheduled to be performed to its 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 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 as the search for proofs of the same linear logic [5] theory extended with subexponentials [14]. Then, we show that a PSN^ν execution for a distributed *NDlog* program derives the same facts as an SN execution for a centralized Datalog program. This property is proved by relating the linear logic proofs specifying PSN^ν computation-runs with the proofs specifying SN computation-runs. 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*, while in Section 3 we review the SN and PSN algorithms, explain the problems of PSN, and informally introduce PSN^ν. Then, in Section 4, we sketch our encodings of SN and PSN^ν in linear logic and in Section 5 we show our main correctness results. Finally in Section 6, we comment on related work and conclude with final remarks in Section 7.

2 Network Datalog Language

In this section, we review the language *Network Datalog* (*NDlog*) [7], which extends Datalog programs, by allowing one to distribute Datalog rules in a network.

2.1 Background: Datalog

We first review some standard definitions of Datalog, following [16]. A *Datalog* program consists of a (finite) set of logic rules and a query. A rule has the form

$\forall \boldsymbol{X}.(h\,\boldsymbol{T}_h \leftarrow b_1\,\boldsymbol{T}_1, \ldots, b_n\,\boldsymbol{T}_n)$, where the commas are interpreted as conjunctions and the symbol $\leftarrow$ as implication; $h\,\boldsymbol{T}_h$ is an atom called the head of the rule; $b_1\,\boldsymbol{T}_1, \ldots, b_n\,\boldsymbol{T}_n$ is a sequence of atoms and function relations called the body; and the $\boldsymbol{T}$s are vectors of variables and ground terms. The variables in $\boldsymbol{X}$ are exactly those appearing in the union of the variables in $\boldsymbol{T}_h$ and $\boldsymbol{T}_i$s. *Function relations* are simple operations such as boolean, or arithmetic (*e.g.* $X_1 < X_2$), or list manipulations operations (*e.g.* $app\ L_1\ L_2\ L_3$). 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[3]. We classify the predicates that do not depend on any other predicates as base predicates, and the remaining predicates as derived predicates. Consider the following non-recursive Datalog program where p, s, and t are a derived predicates and u, q, and r are base predicates:$\{p \leftarrow s, t, r;\ s \leftarrow q;\ t \leftarrow u;\ q \leftarrow;\ u \leftarrow\}$. The set of all the ground atoms that are derivable from this program, called *view*, is the multiset $\{s, t, q, u\}$.

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* program, we provide an example based on a simplified version of the *path-vector* protocol, a standard routing protocol used for 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

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

with cycles, rule `r2` uses function `f_inPath`, where `f_inPath(P,S)` returns true if `S` is in the path vector `P`.

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 different node. To improve performance and eliminate unnecessary communication, we use a *rule localization* [7] 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 rule non-local when the rule head and body have different location specifiers. We use the convention that a non-local rule resides in the same location as its body predicates, and that when the rule is used, the derived head predicate will be *sent* to the appropriate location as specified. For the rest of this paper, we assume that the localization rewrite has been performed.

3 Network Datalog Program Execution

The evaluation of *NDlog* programs uses *pipelined semi-naïve* (PSN) algorithm, which is based on *semi-naïve fixed point* [2] Datalog evaluation mechanism (SN). We provide a brief review of SN algorithm, before describing the PSN extension.

3.1 Semi-Naïve Algorithm

When base predicates are updated, these updates need to be propagated so that the views are consistent with the Datalog rules and current base predicate. 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.

Given a set of insertions, I_k, and deletions, D_k of base predicates, the Algorithm 1 can be used to maintain the view of a Datalog program. First, we create for each rule $\forall \boldsymbol{X}.(h\,\boldsymbol{T}_h \leftarrow b_1\,\boldsymbol{T}_1, \ldots, b_n\,\boldsymbol{T}_n)$ in a Datalog program the following delta insertion and deletion rules:

$$\{\forall \boldsymbol{X}.(\textsc{ins}(h)\,\boldsymbol{T}_h \leftarrow b_1^{\nu}\,\boldsymbol{T}_1, \ldots, b_{i-1}^{\nu}\,\boldsymbol{T}_{i-1}, \varDelta b_i\,\boldsymbol{T}_i, b_{i+1}\,\boldsymbol{T}_{i+1}, \ldots, b_n\,\boldsymbol{T}_n) \mid 1 \leq i \leq n\}$$
$$\{\forall \boldsymbol{X}.(\textsc{del}(h)\,\boldsymbol{T}_h \leftarrow b_1^{\nu}\,\boldsymbol{T}_1, \ldots, b_{i-1}^{\nu}\,\boldsymbol{T}_{i-1}, \varDelta b_i\,\boldsymbol{T}_i, b_{i+1}\,\boldsymbol{T}_{i+1}, \ldots, b_n\,\boldsymbol{T}_n) \mid 1 \leq i \leq n\}$$

Intuitively, given a set of insertions, I_k, and deletions, D_k, of base predicates, the Algorithm 1 uses these rules to incrementally maintain a view as follows: if we are in, say, the $i^{th}+1$ iteration, then the contents of p corresponds to the view of p at iteration $i-1$ and the contents of p^{ν} to the view at iteration i. The $i^{th}+1$ iteration consists of executing the delta-rules for all updates in I_k and D_k, and whenever an insertion or deletion rule is fired, we store the derived tuple in respectively I_k^{ν} and D_k^{ν}. Once all rules have been executed, we update the view accordingly and proceed to a new 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 last lines of the code which use *set-operations*.

Algorithm 1 maintains correctly the view of a Datalog program [6] whenever there is one and only one derivation for any tuple. This limitation is due to the use of set semantics. Other more complicated algorithms are available,

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

```
@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)]                        {}[]                        {}[]
```

Fig. 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.

but formalizing them seems to be a non-trivial task. Moreover, Algorithm 1 captures most of the programs used until now in declarative networking. For instance, we can use it to maintain the datalog program corresponding to the path vector program described above since each `path` tuple is supported by just one derivation.

3.2 Existing Pipelined Semi-naïve Evaluation

In order to maintain incrementally the states of nodes in a distributed setting, Loo *et al.* in [7, 8] proposed PSN. In PSN, each node has a queue of *messages* scheduling insertions and deletions of tuples to the node's local state. A node 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 it is 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 whose view 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 :-
```

Moreover, consider 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. In the final transitions, none of the updates deleting `s` and `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 differently from 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 node 1's queue.

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 node's `1` queue 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 [7, 8], 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 to all the system to reach a fixed point; that message channels are assumed to be 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^{ν}, 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, it also potentially leads to improved performance, since the implementation no longer requires receiver-based network buffers necessary to guarantee in-order delivery of messages.

3.3 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*, called *upd*, 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 *upd*. We process *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, with the invariant that before and after a PSN^{ν}-iteration, the contents in p and in p^{ν} are the same:

pick – One picks (non-deterministically) any update, u, from the bag *upd*, except if the u is a deletion of an atom that is not (yet) in the view. Then, if u is an insertion of predicate p, we add it to the contents of p^{ν}, otherwise if it is a deletion of the same predicate, we delete it from p^ν;

fire – After picking an update, one executes all the delta-rules corresponding to u. If a rule is fired, then we insert the derived tuple into the bag *upd*.

update – Once all delta-rules are executed, we update the view according to u: if u is an insertion or deletion of predicate p, we insert it into or delete it from the contents of p.

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 *upd* are picked using the *pick* command. The result is that the contents in the p^νs are updated with the updates derived in the previous iteration. Hence, the contents of the p^νs correspond exactly to the view at iteration i, while the contents in p correspond exactly to the view at iteration $i-1$, 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 *upd*. After this phase, *upd* contains the updates derived at iteration $i+1$. Finally, in the third phase, one executes eagerly the *update* command which then updates the contents in p to match the contents in p^ν.

Because both algorithms can be explained by using the same basic commands and the same delta-rules, we are able to prove correctness of *PSN*$^\nu$ by showing that for any computation-run of *PSN*$^\nu$, which formally corresponds to a linear logic proof, there is a computation-run of SN, which corresponds to another linear logic proof of the same sequent, and vice-versa.

4 Encoding PSN^ν and SN in Linear Logic with Subexponentials

We choose to use linear logic to specify the operational semantics of PSN^ν or of SN instead of a transition system, because of the following two reasons. First, linear logic is a precise and well established language, used already for both reasoning and specifying semantics of programming languages. Second, linear logic provides us with a finer detail on how data is manipulated, thus opening the possibility to use our encoding to prove the correctness not only of PSN^ν, but also of how it is implemented.

Although the details of the proof system for linear logic with subexponentials are beyond the scope of this paper, in the next sections, we sketch its role for the specification of both algorithms PSN^ν and SN. The details of the encoding can be found in [13].

4.1 Linear Logic and Subexponentials

We review some of linear logic's basic proof theory. *Literals* are either atoms or their negations. The connectives $\otimes$ and $\parr$ and the units 1 and $\bot$ are *multiplicative*; the connectives $\&$ and $\oplus$ and the units $\top$ and 0 are *additive*; $\forall$ and $\exists$ are (first-order) quantifiers; and ! and ? are the *exponentials*. We assume that all formulas are in *negation normal form*, that is, negation has atomic scope.

Due to the exponentials, one can distinguish in linear logic two kinds of formulas: the linear ones whose main connective is not a ? and the unbounded ones whose main connective is a ?. The linear formulas can be seen as resources that can only be used once, while the unbounded formulas as unlimited resources which can be used as many times necessary. This distinction is usually reflected in syntax by using two different contexts in the sequent, one containing only unbounded formulas and another only linear formulas [1]. Such distinction allows one to incorporate structural rules, *i.e.*, weakening and contraction, into the introduction rules of connectives.

However, the exponentials are not canonical [3]. In fact, we can assume the existence of a proof system containing as many exponential-like operators, ($!^l$, $?^l$) called subexponentials [14], as one needs: they may or may not allow contraction and weakening, and are organized in a pre-order ($\preceq$) specifying the entailment relation between operators. In these proof systems the contexts for the subexponentials are denoted by the function $\mathcal{K}$ which maps the set of *subexponential indexes* to multisets of formulas. If l is a subexponential index, we denote by $\mathcal{K}[l]$ the multiset of formulas associated to l by $\mathcal{K}$. Notice that a context $\mathcal{K}[l]$ behaves either like the linear logic's unbounded context or its linear context depending if the index l allows structural rules or not. The preorder $\preceq$ is used to specify the introduction rule of subexponential bangs. As in its corresponding linear logic rule, to introduce a $!^l$ one needs to check if some type of formulas are not present, namely, that there are no formulas in the linear context nor in the contexts of the indexes k such that $l \not\preceq k$.

Following [14], we use subexponential indexes to encode data structures, such as views, in the context of a sequent. Given a set of ground atoms $\mathcal{D}$, representing a view, for each predicate p, we store its view with respect to $\mathcal{D}$ in the contexts of the subexponentials p and p^ν using the functions: $\mathcal{K}_\mathcal{D}[p] = \{p\ [\boldsymbol{t}] \mid p\,\boldsymbol{t} \in \mathcal{D}\}$ and $\mathcal{K}_\mathcal{D}[p^\nu] = \{p^\nu\ [\boldsymbol{t}] \mid p\,\boldsymbol{t} \in \mathcal{D}\}$, where $[\boldsymbol{t}]$ is a list of terms. We encode in a similar fashion updates using the index *upd*, the query using the function *query*, and the encoding of program delta-rules using the index *rules*. In order to keep track of which updates have been used to fire rules from those that have not, we use the indexes *picked*, where we store updates that where picked from the *upd* bag, and *exec*, where we store updates that have been used to fire delta-rules.

To check if the contexts of the indexes in the set $\mathcal{I}$ are all empty, we follow [14] and create a new index $\hat{l}$ such that $\hat{l} \preceq k$ for all indexes, except those in $\mathcal{I}$. Therefore one can only introduce the subexponential bang of $\hat{l}$ if the contexts for the indexes in $\mathcal{I}$ are all empty.

4.2 Focusing and algorithmic specifications

Focused proof systems, first introduced by Andreoli for linear logic [1], provide normal-form proofs for proof search. Inference rules that are not necessarily invertible are classified as positive, and the remaining rules as negative. Using this classification, focused proof systems reduce proof search space by allowing one to combine a sequence of introduction rules of the same polarity into larger derivations, which can be seen as "macro-rules". The backchaining rule in logic programming can be seen as such macro-rule.

In [14], Nigam and Miller propose the focused system for linear logic with subexponentials called SELLF and show how to specify imperative-like programs. Consider for example the linear logic definitions depicted in Figure 2. In a focused system, these definitions are enforced to behave exactly as one would intuitively imagine. The instructions **load** and **unload** insert and delete an element from a context, while **end** is just used to mark the end of a program.

$$
\begin{array}{lll}
\textbf{load}\ \langle t_1, \ldots, t_n \rangle\ l\ prog & \stackrel{\triangle}{=} & ?^{l}(l\, t_1 \cdots t_n) \invamp prog \\
\textbf{unload}\ l\ \langle v_1, \ldots, v_n \rangle\ bprog & \stackrel{\triangle}{=} & (l\, v_1 \cdots v_n)^{\perp} \otimes (bprog\ v_1 \cdots v_n) \\
\textbf{loop}\ l\ kprog\ prog & \stackrel{\triangle}{=} & \exists v_1 \cdots v_n [(l\, v_1 \cdots v_n)^{\perp} \otimes \\
 & & (kprog\ v_1 \cdots v_n)\ (\textbf{loop}\ l\ kprog\ prog)] \oplus !^{\hat{l}}(prog) \\
\textbf{end} & \stackrel{\triangle}{=} & \perp
\end{array}
$$

Fig. 2. Linear logic definitions for the basic instructions.

$$
\begin{array}{lll}
pick & \stackrel{\triangle}{=} & \exists PLU[\textbf{unload}\ upd\ \langle P, L, U \rangle; \textbf{load}\ \langle P, L, U \rangle\ picked \\
 & & [(U = \textsc{ins}) \otimes \textbf{load}\ \langle L \rangle\ P^{\nu} \textbf{end}] \oplus [(U = \textsc{del}) \otimes \textbf{unload}\ \langle L \rangle\ P^{\nu} \textbf{end})]] \\
fire & \stackrel{\triangle}{=} & \exists PLUR[\textbf{unload}\ picked\ \langle P, L, U \rangle; \textbf{unload}\ rules\ \langle P, R, U \rangle; \\
 & & \textbf{load}\ \langle P, L, U \rangle\ exec; \textbf{load}\ \langle L \rangle\ \Delta P; execRules\ R\, (\textbf{unload}\ \Delta P\ \langle L \rangle\ \textbf{end})] \\
update & \stackrel{\triangle}{=} & \exists PLU[\textbf{unload}\ exec\ \langle P, L, U \rangle \\
 & & [(U = \textsc{ins}) \otimes \textbf{load}\ \langle L \rangle\ P\, \textbf{end}] \oplus [(U = \textsc{del}) \otimes \textbf{unload}\ P\ \langle L \rangle\, \textbf{end}]] \\
query & \stackrel{\triangle}{=} & !^{test} \exists SL[\textbf{unload}\ queryLoc\ \langle S, L \rangle\, (\textbf{unload}\ \langle L \rangle\ S \top)]
\end{array}
$$

Fig. 3. Linear logic definitions specifying the basic commands. We elide from specifications the λ symbols and denote formulas of the form $A\ (B\ C)$ as $(A;\ B\ C)$.

In $\textbf{loop}\, l\, kprog\, prog$, we use a continuation passing style specification. It deletes an atom from the context of l and focuses on the logic formula obtained from applying the terms $v_1 \cdots v_n$ and the continuation ($\textbf{loop}\ l\ kprog\ prog$) to $kprog$. The loop ends when the context of l is empty, specified by the use of the $!^{\hat{l}}$, and then continues by introducing the logic formula $prog$.

The definition $move\ S\ R\ K \stackrel{\triangle}{=} \textbf{loop}\, S\, \lambda T \lambda cont_l (\textbf{load}\, \langle T \rangle\, R\, cont_l)\, K$ illustrates the use of these definitions. It moves all the elements from the context S to the context R, and then proceeds with the logic formula K.

4.3 Basic Commands

The linear logic definition for the basic commands described informally in Section 3 are depicted Figure 3. The basic command *fire* is the most elaborate. It starts by unloading an update, $\langle p, l, u \rangle$, that is in *picked*, where p is predicate name, l a list of terms denoting its arguments, and u is either INS or DEL denoting the type of update; then retrieving the corresponding insertion or deletion delta rules r, for the predicate p; loading and unloading l into Δp, in order to execute its delta rules; and finally loading the tuple $\langle p, l, u \rangle$ in the context *exec*, denoting that the delta rules for this update have been executed. The execution of a rule is done by *execRules*, whose definition can be found in the technical report [13]. Intuitively, one traverses the encoding of the body of a rule building in the process a substitution that satisfies all body elements. If a predicate is encountered, one checks among all elements in its view for the ones that can be used to fire the rule; otherwise if a function relation is encountered, one checks if the partial substitution built satisfies the relation. Once a rule is fired, we insert the derived update in *upd*. Notice that *query* is the only command that can finish a proof due to the presence of $\top$ which is reached only after verifying that the query is in the view. The $!^{test}$ specifies that query can only be used when the contexts for

$upd, picked$, and $exec$ are all empty and therefore there are more updates being processed.

We insert these basic commands in a sequent by using the function $\mathcal{K}_{BC}[\infty] = \{!^{-\infty} pick, !^{-\infty} fire, !^{-\infty} update, !^{-\infty} query\}$, where ∞ $(-\infty)$ is the maximal (minimal) index, that is, $l \preceq \infty$ $(-\infty \preceq l)$ for all index l. Since the maximal index allows both contraction and weakening, the basic commands can be used as many times as needed. The purpose of the minimal index is novel. Due to the focusing discipline, the execution of a basic command is atomic, that is, one can only use a basic command when there is no other basic command being introduced.

Given a set of ground atoms $\mathcal{D}$, a Datalog program $\mathcal{P}$, a multiset of updates $\mathcal{U}$, and a ground atom s, the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ is such that its linear context is empty and its subexponential context is $\mathcal{K}_\mathcal{D} \otimes \mathcal{K}_\mathcal{P} \otimes \mathcal{K}_\mathcal{U} \otimes \mathcal{K}_s \otimes \mathcal{K}_{BC}$, where $\mathcal{K}_{BC}$ is the encoding of basic commands, $\mathcal{K}_s$ is the encoding of the query for s, $\mathcal{K}_\mathcal{U}$ is the encoding of updates, $\mathcal{K}_\mathcal{P}$ the encoding of delta-rules, $\mathcal{K}_\mathcal{D}$ the encoding of the view, and $\mathcal{K}_1 \otimes \mathcal{K}_2[l] = \mathcal{K}_1[l] \cup \mathcal{K}_2[l]$ for any l.

5 Correctness

The following definitions specify the proofs that correspond to computation runs of PSN^ν and of SN, called respectively PSN^ν and SN-proofs. The correctness proof goes by showing that if one proof exists then the other must also exist; or in other words, any query that is entailed by using PSN^ν is also entailed by SN and vice-versa.

Definition 1. *An execution of a basic command BC is any focused derivation that introduces a sequent focused on the formula $!^{-\infty}BC$ and whose rules introduce only descendants of $!^{-\infty}BC$. We say that the execution of* pick *(resp.* fire *and* update*) uses u if u is the element unloaded from* upd *(resp.* picked *and* exec*).*

Definition 2. *A derivation is a complete iteration if it can be partitioned into a sequence of executions of* pick*, followed by a sequence of executions of* fire*, and finally a sequence of executions of* update*, such that the multiset of tuples, $\mathcal{T}$, used by the sequence of* pick *executions is the same as used by the sequence of* fire *and* update *executions. A complete iteration is an SN-iteration if $\mathcal{T}$ contains all tuples at the end-sequent that are in $\mathcal{K}[\mathrm{upd}]$. A complete iteration is a PSN^ν-iteration if $\mathcal{T}$ contains only one element.*

Definition 3. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a Datalog program, $\mathcal{U}$ a multiset of updates, and s be a ground atom. We call any focused proof, Ξ, of the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ as a PSN^ν-proof (respectively SN-proof) if it can be partitioned into a sequence of PSN^ν-iterations (respectively SN-iterations) followed by an execution of* query*.*

Theorem 1. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a non-recursive Datalog program, $\mathcal{U}$ be a multiset of updates, and s be a ground atom. There is a PSN^ν-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ iff there is an SN-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$.*

Corollary 1. *For non-recursive programs, a query is entailed by using PSN^ν iff it is entailed by using SN.*

We prove the theorem above by showing that: 1) we can permute the executions of 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. These operations are formalized by the Lemmas 2 and 3 shown in the Appendix. Given a PSN^{ν}-proof, we construct an SN-proof by induction as follows: we use the first operation to permute downwards the PSN^{ν}-iteration that picks any element in the end-sequent's *upd*'s context, then repeat it with its subproof. The resulting proof has all PSN^{ν}-iterations in the same order as in an SN-Proof. We merge them into SN-iterations by applying the second operation repeatedly. For the converse direction, given an SN-proof, we can repeatedly apply the third operation to split SN-iterations and obtain a PSN^{ν}-proof.

While performing these operations, however, it can happen that new rules are fired. In particular, 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 these 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 must end. This is formalized by Lemma 1 in the Appendix.

However, if we can guarantee such termination for PSN^{ν}, then the proof works exactly in the same way. Let us return to our path-vector example, shown in Section 2, which is a recursive 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 a 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.

In literature, there are algorithms that can be used to determine termination of Datalog programs [11]. It seems possible to adapt them to a distributed setting, but this is left out of the scope of this paper. We are also currently investigating larger classes of programs for which PSN^{ν} terminates.

6 Related Work

Navarro *et al.* propose in [12] 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 also yields unsound results.

7 Conclusions

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 using a novel approach: we encode both the SN and PSN^{ν} in linear logic with subexponentials. The correctness result is proven by showing that a proof that encodes a SN evaluation can be transformed to one that encodes a PSN^{ν} evaluation and vice versa. Focused proofs in linear logic give well-defined operational semantics for PSN^{ν}. Furthermore, PSN^{ν} lifts restrictions such as FIFO channels from *NDlog* implementations and leads to significant performance improvements of protocol execution.

This work is part of a bigger effort to formally analyze network protocol implementations [4, 19]. 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.

References

1. J. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
2. I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Prog*, 4(3):259–262, 1987.
3. V. Danos, J. Joinet, and H. Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In *Kurt Gödel Colloquium*, p. 159–171, 1993.
4. Formally Verifiable Networking. `http://netdb.cis.upenn.edu/fvn/`
5. J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
6. A. Gupta, I.Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.
7. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.
8. B. T. 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.
9. B. T. Loo, T. Condie, J.M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.
10. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.
11. I. S. Mumick and O. Shmueli. Finiteness properties of database queries. In *Australian Database Conference*, pages 274–288, 1993.
12. J. A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In *PADL*, pages 76–90, 2009.
13. V. Nigam, L. Jia, A. Wang, B. T. Loo, and A. Scedrov. An operational semantics for network datalog. Extended version available at `http://netdb.cis.upenn.edu/fvn/ndlogsemantics.pdf`
14. V. Nigam and D. Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP*, pages 129–140, 2009.
15. P2: Declarative Networking System. `http://p2.cs.berkeley.edu`
16. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
17. RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`
18. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
19. A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu. Formally Verifiable Networking. In *SIGCOMM HotNets-VIII*, 2009.

8 Appendix

Lemma 1. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a non-recursive Datalog program, s be a ground atom, and $\mathcal{U}$ be a multiset of updates, such that $\langle p, L, \textsc{ins}\rangle$, $\langle p, L, \textsc{del}\rangle \in \mathcal{U}$. Let $\mathcal{U}' = \mathcal{U} \setminus \{\langle p, L, \textsc{ins}\rangle, \langle p, L, \textsc{del}\rangle\}$ be a multiset of updates. Then the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ has a PSN^ν-proof iff the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}', s)$ has a PSN^ν-proof.*

Proof. ($\Rightarrow$) The updates $\langle p, L, \textsc{ins}\rangle, \langle p, L, \textsc{del}\rangle \in \mathcal{U}$ do not really affect the execution of *query*, since for all insertions propagated by the update $\langle p, L, \textsc{ins}\rangle$ there are the same deletions propagated by the update $\langle p, L, \textsc{del}\rangle$. We can construct the a proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}', s)$ by trimming the pieces of derivations in the proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ that depend on these updates. We do so by induction on the number of PSN^ν-iterations. Let Ψ be the set of updates propagated by $\langle p, L, \textsc{ins}\rangle$ and $\langle p, L, \textsc{del}\rangle$. One determines this set by inspection on the proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$. Consider the following representative inductive case where the proof ends with a PSN^ν-iteration of the form:

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathcal{K}_1 : \cdot \Downarrow (upd\, p_1\, L_1\, u)^\perp} \quad \dfrac{\dfrac{\dfrac{\Xi}{\vdash \mathcal{K}_2' : \cdot \Uparrow \cdot}}{\vdash \mathcal{K}_2' : \cdot \Downarrow \mathbf{end}}}{\vdash \mathcal{K}_2 : \cdot \Downarrow prog}}{\dfrac{\vdash \mathcal{K} : \cdot \Downarrow (upd\, p_1\, L_1\, u)^\perp \otimes prog}{\vdash \mathcal{K} : \cdot \Downarrow \mathbf{unload}\, upd\, \langle p_1, L_1, u\rangle\, prog}}}{\vdash \mathcal{K} : \cdot \Downarrow !^{-\infty} pick}$$

If the update $\langle p_1, L_1, u\rangle$ is an update propagated from $\langle p, L, \textsc{ins}\rangle$ or $\langle p, L, \textsc{del}\rangle$, then this derivation is completely deleted. Otherwise, we should not delete the whole derivation, but only the parts in the execution of *fire* that use tuples in the view which come from insertions propagated from $\langle p, L, \textsc{ins}\rangle$. These deletions are also done by induction, but this time on the number of "loops" in *fire*.

Here is a representative inductive case, where in the derivation below the **loop**s are two consecutive occurrences of loops over p_1:

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathcal{K}_1 :\Downarrow (p_1 \boldsymbol{t})^\perp} \quad \dfrac{\dfrac{\Xi}{\vdash \mathcal{K}_2' :\Downarrow \mathbf{loop}\, p_1\, kprog_2\, prog_2}}{\vdash \mathcal{K}_2 :\Downarrow (kprog\ \boldsymbol{t})\ (\mathbf{loop}\ p_1\ kprog\ prog)}}{\vdash \mathcal{K} :\Downarrow (p_1 \boldsymbol{t})^\perp \otimes (kprog\ \boldsymbol{t})\ (\mathbf{loop}\ p_1\ kprog\ prog)}}{\vdash \mathcal{K} :\Downarrow \mathbf{loop}\, p_1\, kprog\, prog}$$

We delete this derivation only if p_1 is of the forms p or p^ν or p^i_{aux}[4] and the update $\langle p, [\boldsymbol{t}], \textsc{ins}\rangle$ is in Ψ. At the same time, we delete all occurrences of the atoms $(upd\, p\, l\, u)$, $(p\, l)$, $(p^\nu\, l)$, and $(p_{aux}\, l)$ such that the update $\langle p, l, u\rangle$ is in Ψ.

[4] As you can see in the technical report, we assume that for each predicate p there are auxiliary subexponential indexes, p^i_{aux}, used to mark the tuples in p which were already traversed.

($\Leftarrow$) Let Ξ be the given proof of the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}', s)$. Moreover, let Ξ_p be the derivation composed of all PSN^ν-iterations in Ξ and Ξ_q be the derivation composed of the *query* execution in Ξ. We can construct a proof of the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ as follows. We add to the context *upd* of all sequents in Ξ_p that are not introduced by an initial rule the updates $\langle p, L, \textsc{ins}\rangle$ and $\langle p, L, \textsc{del}\rangle$. Let Ξ'_p be the resulting derivation. Then the end sequent of Ξ'_p is $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ and its open premise is such that the context of *upd* is composed exactly of the updates $\langle p, L, \textsc{ins}\rangle$ and $\langle p, L, \textsc{del}\rangle$. Now, since the program is non-recursive, it is case that there is a finite sequence of PSN^ν-iterations that computes the updates $\langle p, L, \textsc{ins}\rangle, \langle p, L, \textsc{del}\rangle$ and all the updates propagated by them. Let Ξ_u be the derivation corresponding to such computation[5]. The context of *upd* of Ξ_u's end sequent is the multiset $\{\langle p, L, \textsc{ins}\rangle, \langle p, L, \textsc{del}\rangle\}$, while the same context for its premise is the $\emptyset$. Finally, we can compose the derivations Ξ'_p, Ξ_u, and Ξ_q and construct the proof for $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$.

Lemma 2. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a non-recursive Datalog program, $\mathcal{U}$ be a multiset of updates, such that $u_1, u_2 \in \mathcal{U}$, and s be a ground atom. Let Ξ be a PSN^ν-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ which ends with two PSN^ν-iterations that use u_1 and u_2. Then there is a PSN^ν-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ which ends with two PSN^ν-iterations that use the updates u_2 and u_1.*

Proof. We must consider four different cases, according to the updates u_1 and u_2:

• u_1 and u_2 are both insertions: $\langle p_1, L_1, \textsc{ins}\rangle$ and $\langle p_2, L_2, \textsc{ins}\rangle$. We show that the multiset of firings obtained by first picking $\langle p_2, L_2, \textsc{ins}\rangle$ and then $\langle p_2, L_2, \textsc{ins}\rangle$ is the same as before. Let F_1 be the multiset of firings in the first case and F_2 be the set of firings in the second case. Let $s_1 \in F_1$. If s_1 be a firing obtained in the first PSN^ν-iterations, then it must be the case that $s_1 \in F2$ since the same delta rule is executed. If s_1 is obtained in the second PSN^ν-iteration, then either it did not use the insertion of $\langle p_1, L_1, \textsc{ins}\rangle$, in which case, $s_1 \in F_2$, since the same delta-rule would be executed; or it did use the insertion of $\langle p_1, L_1, \textsc{ins}\rangle$, in which case there is a rule that contains both p_1 and p_2 in the body, and therefore $s_1 \in F_2$ because then its delta rule containing Δp_1 and t in its body is fired. To prove that if $s_2 \in F_2$ then $s_2 \in F_1$ follows the same reasoning.

• u_1 and u_2 are both deletions: $\langle p_1, L_1, \textsc{del}\rangle$ and $\langle p_2, L_2, \textsc{del}\rangle$. The reasoning is similar as in the previous case. Let F_1 be the multiset of firings in the first case and F_2 be the set of firings in the second case.

• u_1 is an insertion and u_2 is a deletion: $\langle p_1, L_1, \textsc{ins}\rangle$ and $\langle p_2, L_2, \textsc{del}\rangle$. Again, we show that the multiset of firings obtained by first picking $\langle p_2, L_2, \textsc{del}\rangle$ and then $\langle p_1, L_1, \textsc{ins}\rangle$ is the same as before. Let F_1 be the multiset of firings in the first case and F_2 be the set of firings in the second case. Let $s_1 = \langle s, L_s, \textsc{ins}\rangle \in F_1$ be an update created in the first PSN^ν-iteration. Then either one did not use L_2

[5] We can search for such computation by just following the algorithm specified in linear logic. We do so by picking any INS update and then the corresponding DEL update. Since in the execution of *fire* we traverse all possible combinations of tuples in the view, it does not really matter in which order we unload elements. Hence, one does not require to backtrack between focusing phases, but just to backtrack inside focusing phases, which is controlled by the size of the "macro-rules".

from p_2, in which case, $s_1 \in F_2$, or one did use L_2 from p_2, in which case it must be that another update $s_1' = \langle s, L_s, \text{DEL}\rangle \in F_2$ is created because a delta rule of the same rule must be fired in the second PSN^ν-iteration. In this case, neither s_1 nor s_1' belong to F_2 because, by inverting the order of picks, no rule is fired. However, from Lemma 1, the resulting sequent is still provable. The reasoning is the same for the case when $s_1 = \langle s, L_s, \text{DEL}\rangle \in F_1$. To show the reverse direction that if $s_2 \in F_2$ then $s_2 \in F_1$, the reasoning is similar to the next case.
• u_1 is a deletion and u_2 is an insertion: $\langle p_1, L_1, \text{DEL}\rangle$ and $\langle p_2, L_2, \text{INS}\rangle$. Once more, we show that the multiset of firings obtained by first picking $\langle p_2, L_2, \text{INS}\rangle$ and then $\langle p_1, L_1, \text{DEL}\rangle$ is the same as before. Let F_1 be the multiset of firings in the first case and F_2 be the set of firings in the second case. Let $s_1 \in\in F_1$, then $s_1 \in F_2$ since the same delta rule must be fired when one picks u_2 before u_1. Now, consider that $s_2 = \langle s, L_s, \text{INS}\rangle \in F_2$ is created in the first PSN^ν-iteration. Then it is created either not using L_2 from p_2, in which case $s_2 \in F_1$, or by using L_2 from p_2, in which case, a it must be that another update $s_2' = \langle s, L_s, \text{DEL}\rangle \in F_2$ is created because a delta rule of the same rule must be fired in the second PSN^ν-iteration. So $s_2, s_2' \notin F_1$. However, again from Lemma 1, the resulting sequent is still provable. The reasoning is the same for when $s_2 = \langle s, L_s, \text{DEL}\rangle \in F_2$.

Lemma 3. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a non-recursive Datalog program, $\mathcal{U}$ be a multiset of updates, such that $\{u\} \cup \mathcal{T} \subseteq \mathcal{U}$, and s be a ground atom. Then there is a proof of the sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ which ends with a complete-iteration that uses the multiset $\mathcal{T}$ followed by a PSN^ν-iteration that uses the update u iff there is a proof of the same sequent that ends with a complete-iteration that uses the multiset $\mathcal{T} \cup \{u\}$.*

Proof. For each direction there are two cases according to the update u to consider. Let F_1 be the multiset of updates created by a complete-iteration, C_1, using $\mathcal{T}$ followed by PSN^ν-iteration, P_1, using u and F_2 be the multiset created by a complete-iteration, C_2, using $\mathcal{T} \cup \{u\}$.
• u is an insertion: $\langle p, L, \text{INS}\rangle$. Let $s_1 \in F_1$ be an update created. If s_1 is created in C_1, then $s_1 \in F_2$ since a delta rule of the same rule is fired in C_2. If s_1 is created in P_1, then either the delta rule that is fired does not use any updates in $\mathcal{T}$, in which case the same delta rule is also fired in C_2, thus $s_1 \in F_2$; or the delta rule use updates in $\mathcal{T}$, in which case there is another delta rule of the same rule that is fired in C_2, namely the one where the delta appears in the right-most position (left-most position) if s_1 insertion (deletion) with respect to the updates used; hence, $s_1 \in F_2$. Now, for the reverse direction, the reasoning is much easier. Let $s_2 \in F_2$ be an update created, by using the update $\langle p, L, \text{INS}\rangle$ then a delta rule of the same rule is fired in P_1; hence $s_2 \in F_1$. Otherwise, the same delta rule is fired in C_1 and therefore $s_2 \in F_1$.
• u is a deletion: $\langle p, L, \text{DEL}\rangle$. Again, let $s_1 \in F_1$ be an update created. If s_1 is created in C_1 not using the tuple L from p, then the same rule is fired in C_2; hence $s_1 \in F_2$. Otherwise, s_1 is created in C_1 using the tuple L from p, then s_1 there is another delta rule of this rule in C_2, hence $s_2 \in F_2$, namely the one where the delta appears in the right-most position (resp. left-most position) if s_1 insertion (resp. deletion) with respect to the updates used. Now, for the reverse direction, the reasoning is similar to the previous case.

Theorem 1. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a non-recursive Datalog program, $\mathcal{U}$ be a multiset of updates, and s be a ground atom. There is a PSN^{ν}-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ iff there is an SN-proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$.*

Proof. ($\Leftarrow$) Given a PSN^{ν}-proof, we construct an SN-proof by induction as follows: use Lemma 2 to permute PSN^{ν}-iteration that picks an element $u \in \mathcal{U}$, then repeat it with its subproof. The resulting proof has all PSN^{ν}-iteration in the same order as in an SN-Proof, but they have to be merged into SN-iterations, which is possible by applying repeatedly Lemma 3. This process terminates since there are finitely many possible updates in a non-recursive program.
($\Rightarrow$) Given an SN-proof, we repeatedly apply Lemma 3 to obtain a PSN^{ν}-proof.