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, 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. 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 [10–13] 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 [20]. 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 [22] 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, these declarative networks result in efficient implementations, as demonstrated in open-source implementations [19, 21].

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 in 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, 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 [7] theory extended with subexponentials [18]. 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* 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. 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 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 later 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 [20]. A *Datalog* program consists of a (finite) set of logic rules and a query. A rule has the form $h(\boldsymbol{t})$ `:-` $b_1(\boldsymbol{t}_1), \ldots, b_n(\boldsymbol{t}_n)$, where the commas are interpreted as conjunctions and the symbol `:-` as implication; $h(\boldsymbol{t})$ 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. We asume the universal quantification of any free variable in a Datalog rule. *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 a 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`}.

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

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

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`.

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 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 *fired*, that is, all of its body elements are true, then 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.

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 [20]. 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 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 $h(\boldsymbol{t})$ `:-` $b_1(\boldsymbol{t}_1), \ldots, b_n(\boldsymbol{t}_n)$ 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:

INS($h(\boldsymbol{t})$) `:-` $b_1^{\nu}(\boldsymbol{t}_1), \ldots, b_{i-1}^{\nu}(\boldsymbol{t}_{i-1}), \Delta b_i(\boldsymbol{t}_i), b_{i+1}(\boldsymbol{t}_{i+1}), \ldots, b_n(\boldsymbol{t}_n)$

DEL($h(\boldsymbol{t})$) `:-` $b_1^{\nu}(\boldsymbol{t}_1), \ldots, b_{i-1}^{\nu}(\boldsymbol{t}_{i-1}), \Delta b_i(\boldsymbol{t}_i), b_{i+1}(\boldsymbol{t}_{i+1}), \ldots, b_n(\boldsymbol{t}_n)$

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 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, and whenever an insertion or deletion rule is fired, we store the derived tuple in I_k^ν and D_k^ν respectively. 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. However, 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 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 (`@`).

```
@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.

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 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 [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 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*$^{\nu}$, 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 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.

3.2 New Pipelined Semi-naïve Evaluation

At a high-level, *PSN*$^{\nu}$ works as follows: Instead of using queues to store unprocessed updates, we use a single *bag*, denoted as *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*$^{\nu}$-iteration is completed by executing in a sequence the following three basic commands, which preserve the invariant that before and after a *PSN*$^{\nu}$-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 *upd*, 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, we insert the corresponding p^ν to the ν table, otherwise if it is a deletion of the same predicate, we delete 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 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 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 *upd* 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 *upd*. After this phase, *upd* 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 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.

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 [4]. In fact, we can assume the existence of a proof system containing as many exponential-like operators, $(!^l, ?^l)$ called subexponentials [18], 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. Now, instead of only two contexts as in linear logic, sequents for such proof systems with subexponentials have besides the linear context Γ as many contexts as needed, In these proof systems the contexts for the subexponentials are denoted by the function $\mathcal{K}$, called *subexponential context*, 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 [18], 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 [18] 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

$$\frac{}{\vdash \mathcal{K} : \Gamma \Uparrow L, \top}\ [\top] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A \quad \vdash \mathcal{K} : \Gamma \Uparrow L, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \mathbin{\&} B}\ [\&] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, \bot}\ [\bot]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A\{c/x\}}{\vdash \mathcal{K} : \Gamma \Uparrow L, \forall x\, A}\ [\forall] \qquad \frac{\vdash \mathcal{K} +_l A : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, ?^l A}\ [?^l] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \mathbin{\invamp} B}\ [\invamp]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow A_i}{\vdash \mathcal{K} : \Gamma \Downarrow A_1 \oplus A_2}\ [\oplus_i] \qquad \frac{\vdash \mathcal{K}_1 : \Gamma \Downarrow A \quad \vdash \mathcal{K}_2 : \Delta \Downarrow B}{\vdash \mathcal{K}_1 \otimes \mathcal{K}_2 : \Gamma, \Delta \Downarrow A \otimes B}\ [\otimes],\ \text{provided } (\mathcal{K}_1 = \mathcal{K}_2)\mid_{I \setminus \mathcal{B}}$$

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow 1}\ [1],\ \text{provided } \mathcal{K}[I \setminus \mathcal{B}] = \emptyset \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow A\{t/x\}}{\vdash \mathcal{K} : \Gamma \Downarrow \exists x\, A}\ [\exists]$$

$$\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l A}\ [!^l],\ \text{provided } \mathcal{K}[\{x \mid l \not\preceq x \wedge x \in \mathcal{B}\}] = \emptyset$$

$$\frac{}{\vdash \mathcal{K} : \Gamma \Downarrow A_p}\ [I],\ \text{provided } A_p^\perp \in (\Gamma \cup \mathcal{K}[I]) \text{ and } (\Gamma \cup \mathcal{K}[\mathcal{B}]) \subseteq \{A_p^\perp\}$$

$$\frac{\vdash \mathcal{K} +_l P : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l],\ \text{provided } l \in I \setminus \mathcal{B} \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l],\ \text{provided } l \in \mathcal{B}$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} : \Gamma, P \Uparrow \cdot}\ [D_1] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow N}{\vdash \mathcal{K} : \Gamma \Downarrow N}\ [R\Downarrow] \qquad \frac{\vdash \mathcal{K} : \Gamma, S \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, S}\ [R\Uparrow]$$

Fig. 2. The focused linear logic system SELLF$_\Sigma$, where $\Sigma = \langle I, \preceq, \mathcal{B}\rangle$. Here, A_p is a positive literal; S is a positive formula or a literal; P is a not a negative polarity literal; and N is a negative formula.

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow B\theta}{\vdash \mathcal{K} : \Gamma \Downarrow p\,\bar{t}}\ [def\Downarrow] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, B\theta}{\vdash \mathcal{K} : \Gamma \Uparrow L, p\,\bar{t}}\ [def\Uparrow]$$

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow t = t}\ [=] \qquad \frac{\{\vdash \mathcal{K}\theta : \Gamma\theta \Uparrow \Delta\theta : \theta \in csu(s,t)\}}{\vdash \mathcal{K} : \Gamma \Uparrow \Delta, s = t}\ [\neq]$$

Fig. 3. Rules for definitions and equalities. In the definition rules, $p\,\bar{t} = H\theta$ and $\forall \boldsymbol{x}[H \triangleq B]$ is a definition. In the equalities rules, $\mathcal{K}\theta[i] = \mathcal{K}[i]\theta$ for all $i \in I$ and $csu(s,t)$ is the complete set of unifiers of s and t.

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" that introduce synthetic connectives. The backchaining rule in logic programming can be seen as such macro-rule.

In [18], Nigam and Miller propose the focused system for linear logic with subexponentials called SELLF$_\Sigma$, where Σ is a tuple $\langle I, \preceq, \mathcal{B}\rangle$ such that $\langle I, \preceq\rangle$ is a preorder and $\mathcal{B} \subseteq I$. Intuitively, I is set of subexponential indexes and its subset $\mathcal{B}$ specifies the subexponentials that do not allow for contraction nor for weakening.

We usually elide the subexponential context whenever it is clear from the context. In order to introduce SELLF, we first classify formulas whose main connective is $\exists, \otimes, \oplus, 1$, and the subexponential bang, and positive literals as positive. The remaining formulas are classified as negative. SELLF is a straightforward generalization of Andreoli's sytem. As in the original presentation for linear logic, there are two sequents: one with the $\Uparrow$ which belongs to the negative phase, and another with the $\Downarrow$ which belongs to the positive. Each sequent has two contexts to the left of the arrow of the form $\mathcal{K} : \Gamma$. The multiset Γ is the linear context that collects the formulas whose main connective is not a subexponential question-mark, and $\mathcal{K}$ is an indexed function from the set I of subexponential indexes to multiset of formulas. Given a subexponential signature $\langle I, \preceq, \mathcal{B}\rangle$, we specify the following operations over these contexts:

- $(\mathcal{K}_1 \otimes \mathcal{K}_2)[i] = \begin{cases} \mathcal{K}_1[i] \cup \mathcal{K}_2[i] & \text{if } i \notin \mathcal{C} \\ \mathcal{K}_1[i] & \text{if } i \in \mathcal{C} \cap \mathcal{W} \end{cases}$ • $\mathcal{K}[\mathcal{S}] = \bigcup\{\mathcal{K}[i] \mid i \in \mathcal{S}\}$
- $(\mathcal{K} +_l A)[i] = \begin{cases} \mathcal{K}[i] \cup \{A\} & \text{if } i = l \\ \mathcal{K}[i] & \text{otherwise} \end{cases}$ • $\mathcal{K} \leq_i [l] = \begin{cases} \mathcal{K}[l] & \text{if } i \preceq l \\ \emptyset & \text{if } i \npreceq l \end{cases}$
- $(\mathcal{K}_1 \star \mathcal{K}_2) \mid_{\mathcal{S}}$ is true if and only if $(\mathcal{K}_1[j] \star \mathcal{K}_2[j])$

where $i \in I$, $j \in \mathcal{S}$, $\mathcal{S} \subseteq I$, and $\star \in \{=, \subset, \subseteq\}$.

To illustrate how algorithmic specifications can be specified in SELLF, consider the following linear logic definitions:

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

In a focused system, these definitions are enforced to behave as follows [18]: The definition **load** $\langle \bar{t}\rangle$ l $prog$ inserts in the context l an atom whose terms are $\bar{t}$ and proceeds introducing the logic formula *prog*. The second definition, **unload** $l\, \langle \bar{v}\rangle\ bprog$, deletes an atom from the context l and procedes introducing the logic formula obtained by applying the terms $\bar{v}$ to *bprog*. We use a continuation passing style specification by using the definition **loop** l *kprog prog*. It intuitively 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 (**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*. Finally, the definition **end** is just used to mark the end of a program instruction.

The definition $move\ S\ R\ K \stackrel{\Delta}{=} \textbf{loop}\ S\ \lambda T \lambda cont_l(\textbf{load}\ \langle T\rangle\ R\ cont_l)\ K$ 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 Encoding views, updates, and queries

Given a set of ground atoms, $\mathcal{D}$, specifying a view, we encode it into the sequent of a proof by using the following function defined over predicate names p:

$$\mathcal{K}_{\mathcal{D}}[p] = \{p\ [t_1, \ldots, t_n] \mid p\, t_1, \ldots, t_n \in \mathcal{D}\} \quad \text{and} \quad \mathcal{K}_{\mathcal{D}}[p^{\nu}] = \{p^{\nu}\ [t_1, \ldots, t_n] \mid p\, t_1, \ldots, t_n \in \mathcal{D}\},$$

where $[t_1, \ldots, t_n]$ is a list of terms.

A multiset of updates, $\mathcal{U}$, is a multiset tuples of the form $\langle p, L, \text{INS}\rangle$ and $\langle p, L, \text{DEL}\rangle$, where $L = [t_1, \ldots, t_n]$ is a list of ground terms and p a predicate name. These tuples denote that the ground atom $p\, t_1 \cdots t_n$ has to be respectively inserted or deleted from the view. A query is a ground atom, $s = q\ t_1 \cdots t_n$, for which we would like to determine its membership in the database after all updates have been propagated. They are encoded int SELLF in the context *upd* and *query* as follows

$$\mathcal{K}_{\mathcal{U}}[upd] = \{upd\, p\, L\, u \mid \langle p, L, u\rangle \in \mathcal{U}\} \quad \text{and} \quad \mathcal{K}_s[query] = \{query\ q\, [t_1, \ldots, t_n]\}.$$

4.4 Encoding (delta) rules

A Datalog delta-rule, $\forall X_1, \ldots, X_m[U(p)\,\boldsymbol{T_p} \leftarrow s_1\,\boldsymbol{T_1}, \ldots, s_n\,\boldsymbol{T_n}]$, is encoded as the tuple $\langle m, \mathit{head}, \mathit{body}\rangle$, where the natural number m specifies the number of universally quantified variables in the rule; the tuple $\mathit{head} = \langle p, U, \mathcal{N}\rangle$ specifies the predicate name of the head of the rule (p), if the rule is an insertion or deletion rule (U), and the list of natural numbers and ground terms, $\mathcal{N}$, denotes the bounded variables and terms used in head of the rule; and finally the list of tuples $\mathit{body} = [\langle B_1, s_1, \mathcal{N}_1\rangle, \ldots, \langle B_n, s_n, \mathcal{N}_n\rangle]$ encodes similarly the body of the rule, where for the i^{th} element in the body of the rule, B_i specifies if it is a predicate ($B = \mathit{pr}$), or a function relation ($B = \mathit{fu}$); s_i is the name of the element; and $\mathcal{N}_i$ is the list of natural numbers and terms specifying the bounded variables and terms used.

As an example, the insertion rule $\forall XYZ[(\textsc{ins}(p)\,a\,Z) \leftarrow (\Delta s\,Y\,X), (\mathit{leq}\,z\,Z)]$ is encoded as the tuple $\langle 3, \mathit{head}, \mathit{body}\rangle$, where the first component, 3, corresponds to the number of bounded variables, X, Y, and Z, in the rule; head is the tuple $\langle p, \textsc{ins}, [a, 3]\rangle$ specifying that the head of the rule takes as arguments the term a and third bounded variable Z; and body is the list $[\langle \mathit{pr}, \Delta s, [2, 1]\rangle, \langle \mathit{fu}, \mathit{leq}, [z, 3]\rangle]$ specifying that the first body element is a predicate and the second a function relation.

Given a Datalog program $\mathcal{P}$, let $\mathcal{P}_\Delta$ be the set of insertion and deletions rules obtained from $\mathcal{P}$. Let $R(\mathcal{P}_\Delta, p, U) = [R^p_1, \ldots, R^p_n]$ be any list of the encodings of different insertion, if $U = \textsc{ins}$, or deletion, if $U = \textsc{del}$, delta rules in $\mathcal{P}_\Delta$ with Δp in its body. Then we encode $\mathcal{P}$ in SELLF by storing the tuples $\langle p, R(\mathcal{P}_\Delta, p, U), U\rangle$ in the context rules, as follows:

$$\mathcal{K}_\mathcal{P}[\mathit{rules}] = \{\mathit{rules}\; p\, R(\mathcal{P}_\Delta, p, U)\, U \mid p \text{ is a predicate name.}\}$$

Depending on the number of updates propagated, rules can be used several times, and therefore the index rules allows contraction and weakening differently from the indexes used for storing views, updates, and the query.

4.5 Basic Commands

The linear logic definition for the basic commands described informally in Section 3 are depicted Figure 4. The basic command *pick* is specified by unloading (non-deterministically) any update tuple, $\langle p, l, u\rangle$, from the context of *upd* and then loading this tuple in the context of *picked*, denoting that its corresponding delta rules should be executed. We also update the context p^ν according to the type u of the update, namely, we remove (reps. insert) the tuple l if u is a deletion (resp. insertion). The basic command *fire* is the most elaborate. It starts by unloading an updated, $\langle p, l, u\rangle$, that is in *picked*; then retrieving the corresponding insertion or deletion delta rules, r, for the predicate p; loading and unloading l into Δt, 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.

In order to execute a rule, we need to traverse all possible combinations of tuples that are currently in the view of predicates appearing in rule's body. First, we start create by using *createSubs* a substitution, represented by a list, S, of the same length, M, as the number of universally quantified variables in the rule. Initially, all elements of S are *unk* denoting that no substitution is assigned to a particular variable. While traversing the views of a rule's body, this substitution is used either to check if an entry can be used to fire this rule or it is replaced by a more specific substitution which contains less *unk* elements. This traversing is done in the definitions of *execAux*: the definition for $\mathit{execAux}\, S\, \langle P, U, \mathcal{N}\rangle\, []\, K$ is used when all elements of the body have been traversed and therefore one has a satisfying substitution list S. We add to the context *upd* the insertion or deletion update, according to U, of the list of terms T_L found by picking the respective terms from the substitution list S by using the predicate *fSlist*.

The definition for $\mathit{execAux}\, S\, H_d\, [\langle \mathit{pr}, P, \mathcal{N}\rangle | B_d]\, K$ is used when we are traversing a body element that is a predicate. We first find an auxiliary location P^i_{aux} of P that is empty and copy all contents of P to it. Then loop through all elements, T_L, in P^i_{aux}, updating the substitution list S to a more specific substitution S_u if the tuple T_L does not conflict with the terms currently used in S, or otherwise ending the loop and trying to find a different combination of tuples that satisfy the rule's body. Given a list of terms, T_L, a list of natural numbers, $\mathcal{N}$, and a substitution list, S, we use the auxiliary predicates depicted in Figure 6 to

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

Fig. 4. Linear logic definitions specifying the basic commands. We elide from specifications the λ symbols and denote formulas of the form $A\ (B\ C)$ as $(A;\ B\ C)$.

$$\begin{array}{lcl}
\mathit{execRules}\,[R \mid L]\,K & \triangleq & \mathit{execute}\,R\,(\mathit{execRules}\,L\,K) \\
\mathit{execRules}\,[]\ K & \triangleq & K \\
\mathit{execute}\,\langle M, H_d, B_d\rangle\,K & \triangleq & \exists S.[\mathit{createSubs}\,S\,M \otimes (\mathit{execAux}\,S\,H_d\,B_d\,K)] \\
\mathit{execAux}\,S\,\langle P, U, \mathcal{N}\rangle\,[]\,K & \triangleq & \exists T_L.[\mathit{fSlist}\,\mathcal{N}\,S\,T_L \otimes \textbf{load}\,\langle P, T_L, U\rangle\ \mathit{upd}\,K] \\
\mathit{execAux}\,S\,H_d\,[\langle \mathrm{pr}, P, \mathcal{N}\rangle | B_d]\,K & \triangleq & \exists i.[\mathit{selEmptyLoc}\,P^i_{\mathrm{aux}} \\
 & & \mathit{copy}\ P\ P^i_{\mathrm{aux}} \\
 & & \textbf{loop}\ P^i_{\mathrm{aux}}\ \lambda T_L \lambda \mathit{cont}_l \\
 & & \quad \exists S_u.[\mathit{findUnif}\ \mathcal{N}\ T_L\ S\ S_u \otimes \\
 & & \quad (!^{\infty}(S_u \neq \mathit{no}) \otimes \mathit{execAux}\,S_u\,H_d\,B_d\,\mathit{cont}_l \\
 & & \quad \oplus \\
 & & \quad (S_u = \mathit{no}) \otimes \mathit{cont}_l)]) \\
 & & K \\
\mathit{execAux}\,S\,H_d\,[\langle \mathrm{fu}, F, \mathcal{N}\rangle | B_d]\,K & \triangleq & (F = \mathit{leq}) \otimes \\
 & & \quad \exists T1T2.[(\mathit{fSlist}\,\mathcal{N}\,S\,[T_1, T_2]) \otimes \\
 & & \quad [(\mathit{leq}\,T_1\,T_2) \otimes \mathit{execAux}\,S\,H_d\,B_d\,K] \\
 & & \quad \oplus \\
 & & \quad [(\mathit{gr}\,T_1\,T_2) \otimes K]] \\
 & & \oplus \\
 & & \ldots
\end{array}$$

Fig. 5. Main definitions for rule execution. The predicate $\mathit{fSlist}\,\mathcal{N}\,S\,T_L$ constructs the list of ground terms T_L by replacing the numbers in $\mathcal{N}$ by the element in S appearing at the correspondent position. The predicate *selEmptyLoc* selects any auxiliary location P^i_{aux} for the predicate P that contains no element. Finally, the predicate *copy* just copies the elements from the context of on index to the context of another index. Here we also assume that there are enough auxiliary indexes for each predicate.

$$\begin{array}{lcl}
\mathit{findUnif}\ \mathcal{N}\ T_L\ S\ S_f & \triangleq & \exists S_r S_u.[\mathit{fSlist}\,\mathcal{N}\,S\,S_r \otimes \mathit{fUnAux}\,T_L\,S_r\,S_u \otimes \\
 & & (S_r = \mathit{no} \otimes S_f = \mathit{no}) \oplus \\
 & & (!^{\infty} S_r \neq \mathit{no} \otimes \mathit{updSubst}\,\mathcal{N}\,S\,S_u\,S_f)] \\
\mathit{fUnAux}\ [T \mid T_L]\,[S \mid S_L]\,[T \mid S_r] & \triangleq & [!^{\infty}(T \neq \mathit{unk}) \otimes (T = S) \otimes \mathit{fUnAux}\ T_L\,S_L\,S_r] \oplus \\
 & & [T = \mathit{unk} \otimes \mathit{fUnAux}\ T_L\,S_L\,S_r] \\
\mathit{fUnAux}\ [T \mid T_L]\,[S \mid S_L]\,\mathit{no} & \triangleq & !^{\infty}(T \neq \mathit{unk}) \otimes !^{\infty}(T \neq S) \\
\mathit{createSubs}\,z\,[] & \triangleq & 1 \\
\mathit{createSubs}\,(s\ M)\,[\mathit{unk}|L_1] & \triangleq & \mathit{createSubs}\,M\,L_1
\end{array}$$

Fig. 6. Auxiliary definitions used in the process of checking if two terms are unifiable and finding a unifier. The predicate *updSubst* just replaces the elements of S appearing at the positions specified in the list $\mathcal{N}$ by the terms in S_u appearing at the same positions resulting in the list S_f. We assume that the constant *unk* is a new constant not appearing in the Datalog program's alphabet.

both update S to a more specific substitution if there is no conflict between the terms in T_L and the terms in S that appear in the positions specified in $\mathcal{N}$; or otherwise returns *no*.

Finally, the definition for $execAux\, S\, H_d\, [\langle fu, F, \Lambda_L\rangle | B_d]\, K$ just checks if the arguments of the function F, given the substitution S, maps F to true. Here we show only the case when F is the function less or equal which are specified in logic as usual. If it is the case, then one continues to traverse the body, otherwise one must pick a different combination of tuples.

The definition for the basic command *update* just updates the contents of a predicate whose delta rules have been executed. Finally, the basic command *query* can only be used when the contexts for *upd*, *picked*, and *exec* are empty, which is specified by the use of the $!^{test}$. It is also 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.

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 ∞ (resp. $-\infty$) is the maximal (resp. 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. It ensures that 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. Due to the focusing discipline, whenever we use a basic command, that is, focus on one of the formulas above, we need to immediately introduce the $!^{-\infty}$, which is only applicable if the linear context is empty, that is, when there are no other basic commands being introduced. This intuition is formalized by Proposition 3.

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 defined as:

$$\vdash \mathcal{K}_{\mathcal{D}} \otimes \mathcal{K}_{\mathcal{P}} \otimes \mathcal{K}_{\mathcal{U}} \otimes \mathcal{K}_s \otimes \mathcal{K}_{BC} : \cdot \Uparrow \cdot,$$

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, and $\mathcal{K}_{\mathcal{D}}$ the encoding of the view.

Definition 1. *Let α be a rule with active formula F and G be a formula produced by α. Then we say that G is the* immediate descendant *of F and that all other formulas appearing in the premises of α are* immediate descendants *of the same formula appearing in α's conclusion. In a derivation, the transitive and reflexive closure of the* immediate descendant *relation specifies the* descendant *relation.*

Definition 2. *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 $!^{-\infty}BC$. A complete execution of a basic command BC is an execution whose premises do not contain any descendants of 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*).*

Proposition 1. *Any complete execution of any basic command has only one open premise.*

Proof Induction on the height of derivations. □

Proposition 2. *Let $\mathcal{D}$ be a set of ground atoms, $\mathcal{P}$ be a Datalog program, and $U = \langle q, l, u\rangle$ be an update. Let Ξ be any complete execution of* fire *using U, whose premise is a fire-ready sequent with subexponential context $\mathcal{K}_e$ and premise with subexponential context $\mathcal{K}_p$. Then $\mathcal{K}_p[i] = \mathcal{K}_e[i]$ for all indexes i different from* upd*,* picked*, and* exec*, $\mathcal{K}_p[upd] = \mathcal{K}_e[upd] \cup \mathcal{F}(\mathcal{D}, \mathcal{P}, U)$, $\mathcal{K}_e[picked] = \mathcal{K}_p[picked] \cup \{picked\, q\, l\, u\}$, and $\mathcal{K}_p[exec] = \mathcal{K}_e[exec] \cup \{exec\, q\, l\, u\}$.*

Proof Induction on the height of derivations. It is easy to check that the only context that has its contents changes is *upd*. All other contexts are used as auxiliary contexts which are empty in both the premise and endsequent of Ξ.

For soundness, we just inspect Ξ and extract the elements in $\mathcal{D}$ that can fire a rule...
□

Proposition 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. Then any focused proof of $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$ can be partitioned into executions of basic commands. Moreover, the top-most execution is of the command* query*.*

Proof By induction on the height of proofs. Because of the $!^{-\infty}$ appearing before the encoding of basic instructions in the end sequent $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$, one is enforced to introduce the defined atoms that do not have a question-mark as main connective before focusing on another basic command.

Since the only command that can close a proof is *query*, the top-most execution of the proof has to be of it. □

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 3. *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 4. *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. In this case, we say that the complete-iteration uses the multiset $\mathcal{T}$. A complete iteration is an SN-iteration if $\mathcal{T}$ contains all tuples at the end-sequent that are in $\mathcal{K}[\text{upd}]$. A complete iteration is a PSN^ν-iteration if $\mathcal{T}$ contains only one element.*

Definition 5. *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*.*

The following lemma states that given a non-recursive program, then conflicting updates, that is, updates inserting and deleting the same tuple, do not interfere in the final output of the PSN^ν computation. The restriction to non-recursive programs is because we cannot guarantee in general termination of PSN^ν in the presence of conflicting updates. For example, the same program used above to show divergence of PSN would also make PSN^ν diverge. However, as we argue later, if such a termination is guaranteed then the proof works in exactly the same way.

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 $\langle p, L, \text{INS}\rangle$, $\langle p, L, \text{DEL}\rangle \in \mathcal{U}$. Let $\mathcal{U}' = \mathcal{U} \setminus \{\langle p, L, \text{INS}\rangle, \langle p, L, \text{DEL}\rangle\}$ 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 $\langle p, L, \text{INS}\rangle, \langle p, L, \text{DEL}\rangle \in \mathcal{U}$ do not really affect the execution of *query*, since for all insertions propagated by the update $\langle p, L, \text{INS}\rangle$ there are the same deletions propagated by the update $\langle p, L, \text{DEL}\rangle$. 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 $\langle p, L, \text{INS}\rangle$ and $\langle p, L, \text{DEL}\rangle$. 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:

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

If the update $\langle p_1, L_1, u\rangle$ is an update propagated from $\langle p, L, \text{INS}\rangle$ or $\langle p, L, \text{DEL}\rangle$, 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 $\langle p, L, \text{INS}\rangle$. 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:

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

We delete this derivation only if p_1 is of the forms p or p^ν or p^i_{aux} and the update $\langle p, [\boldsymbol{t}], \text{INS}\rangle$ 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.

($\Leftarrow$) 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 $\langle p, L, \text{INS}\rangle$ and $\langle p, L, \text{DEL}\rangle$. 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 $\langle p, L, \text{INS}\rangle$ and $\langle p, L, \text{DEL}\rangle$. Now, since the program is non-recursive, it is case that there is a finite sequence of PSN^ν-iterations that computes the updates $\langle p, L, \text{INS}\rangle, \langle p, L, \text{DEL}\rangle$ and all the updates propagated by them. Let Ξ_u be the derivation corresponding to such computation[2]. The context of *upd* of Ξ_u's end sequent is the multiset $\{\langle p, L, \text{INS}\rangle, \langle p, L, \text{DEL}\rangle\}$, 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 $\mathcal{S}(\mathcal{D}, \mathcal{P}, \mathcal{U}, s)$. $\square$

The following lemma states that we can permute the order of how we pick updates to execute PSN^ν-iterations. 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. Provability is not lost as stated in the previous lemma.

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.*

[2] 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".

Proof We must consider four different cases, according to the updates u_1 and u_2:
• u_1 and u_2 are both insertions: $\langle p_1, L_1, \text{INS}\rangle$ and $\langle p_2, L_2, \text{INS}\rangle$. We show that the multiset of firings obtained by first picking $\langle p_2, L_2, \text{INS}\rangle$ and then $\langle p_1, L_1, \text{INS}\rangle$ 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 is a firing obtained in the first PSN^ν-iteration, 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 $\langle p_1, L_1, \text{INS}\rangle$, in which case, $s_1 \in F_2$, since the same delta-rule would be executed; or it did use the insertion of $\langle p_1, L_1, \text{INS}\rangle$, 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: $\langle p_1, L_1, \text{DEL}\rangle$ and $\langle p_2, L_2, \text{DEL}\rangle$. 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: $\langle p_1, L_1, \text{INS}\rangle$ and $\langle p_2, L_2, \text{DEL}\rangle$. Again, we show that the multiset of firings obtained by first picking $\langle p_2, L_2, \text{DEL}\rangle$ and then $\langle p_1, L_1, \text{INS}\rangle$ 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 $s_1 = \langle s, L_s, \text{INS}\rangle \in F_1$ be an update created in the first PSN^ν-iteration. Then either one did not use L_2 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 $s_1 \in F_1$, 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 $s_2 = \langle s, L_s, \text{DEL}\rangle \in F_2$.
□

The following lemma states that we can merge a complete-iteration and a PSN^ν-iteration into a larger complete-iteration, and conversely we can split a larger complete-iteration into a smaller complete-iteration and a PSN^ν-iteration.

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. $\square$

The following theorem uses the operations on proofs formalized in the lemmas above to transform PSN^ν-proofs into SN-proofs and vice-versa, proving hence the correctness of PSN^ν.

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. $\square$

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

In the theorem above, we restricted ourselves to non-recursive programs. The reason for this restriction was just because of issues involvong termination in the presence of conflicting updates. If we can guarantee such termination for PSN^ν, however, 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 [16]. 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.* 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.

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 clients, 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 located resources. Here on the other hand, we encode located resources in the level of propositions by using subexponentials.

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 [5, 23]. 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.

References

1. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. 2(3):297–347, 1992.
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 and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP*, pages 129–140, 2009.
19. P2: Declarative Networking System. `http://p2.cs.berkeley.edu`.
20. Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
21. RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`.
22. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
23. Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally Verifiable Networking. In *SIGCOMM HotNets-VIII*, 2009.