Provably Correct Distributed Provenance Compression

Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Loo and Wenchao Zhou

April 1, 2017

CMU-CyLab-17-001

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Provably Correct Distributed Provenance Compression

(Extended Technical Report)

Chen Chen
University of Pennsylvania
chenche@seas.upenn.edu

Harshal Tushar Lehri
University of Pennsylvania
halehri@seas.upenn.edu

Lay Kuan Loh
Carnegie Mellon University
lkloh@cmu.edu

Anupam Alur
University of Pennsylvania
aalur@seas.upenn.edu

Limin Jia
Carnegie Mellon University
liminjia@cmu.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

ABSTRACT

Network provenance, which records the execution history of network events as meta-data, is becoming increasingly important for network accountability and failure diagnosis. For example, network provenance may be used to trace the path that a message traversed in a network, or to reveal how a particular routing entry was derived and the parties involved in its derivation. A challenge when storing the provenance of a live network is that the large number of the arriving messages may incur substantial storage overhead. In this paper, we explore techniques to dynamically compress distributed provenance stored at scale. Logically, the compression is achieved by grouping equivalent provenance trees and maintaining only one concrete copy for each equivalence class. To efficiently identify equivalent provenance, we (1) introduce distributed event-based linear programs (DELP) to specify distributed network applications, and (2) statically analyze DELPs to allow for quick detection of provenance equivalence at runtime. Our experimental results demonstrate that our approach leads to significant storage reduction and query latency improvement over alternative approaches.

Keywords

Provenance; distributed systems; storage; static analysis

1. INTRODUCTION

Network administrators require the capability to identify the root causes of performance slowdowns in data centers or across wide-area networks, and also to determine the sources of security attacks. Such capabilities often utilize *network provenance*, which allows the user to issue queries over network meta-data about the execution history. In recent years, network provenance has been successfully applied to various network settings, resulting in proposals for distributed provenance [27], secure network provenance [25], distributed time-aware provenance [26] and negative provenance [22]. These proposals demonstrate that database-style declarative queries can be used for maintaining and querying distributed provenance at scale. Moreover, a wide range of forensic analysis work (e.g.[4, 21]) for determining and fixing the root causes of misconfigurations, errors and attacks have used network provenance as their underlying infrastructure.

One of the main drawbacks of the existing techniques is their storage overhead. Network provenance has to be incrementally maintained as network events occur. This is particularly challenging for the *data plane* of networks that deals with frequent and high-volume data packets. When there are streams of incoming packet events, the provenance information can become prohibitively large. While there is prior work on provenance compression in the database literature [3], the work was not designed for distributed settings. Our paper's contributions are:

System Model. We propose a new network programming model, called *distributed event-based linear programs* (DELP), which is a restricted variant of the Network Datalog [11] language in declarative networking. Each DELP program is composed of a set of rules triggered by events, and executes until a fixpoint is reached. Unlike traditional event-condition-action rules, DELP has the option of *slow changing tuples*, which do not change their values while a distributed fixpoint computation is happening. We show, through two example applications (packet forwarding and DNS resolution), that this model is general enough to cover a wide range of network applications.

Distributed Provenance Compression. Based on the DELP model, we propose two techniques to store provenance information efficiently. Our first technique relies on materializing only the tuples that the administrators are interested in. We propose a distributed querying technique that can reconstruct the entire provenance tree from the reduced provenance information that is maintained. Our second technique combines multiple provenance trees together, based on a notion of *equivalence class* that groups different DELP rule firing instances together by virtue of the fact that they share similar derivation structures.

Implementation and Evaluation. We implement a prototype of DELP based on the RapidNet declarative networking engine [13]. We enhance RapidNet to include a rule rewrite engine that maintains provenance at runtime. Provenance queries are implemented as distributed recursive queries over the maintained provenance information. We deploy and evaluate DELP on packet forwarding and DNS lookups, and the performance results show that the com-

pression techniques result in orders of magnitude reduction in storage, significant reduction in query lantency, and adds only negligible overhead to the runtime performance of each monitored network application.

2. BACKGROUND

We first provide an introduction to *Network Datalog* (NDLog) [11], a declarative networking programming language we use to model network applications in the distributed system, then we introduce the concept of distributed network provenance [27, 26].

2.1 Network Datalog

$$\begin{array}{lll} r1 & \mathsf{packet}(@N,S,D,DT) & :- \ \mathsf{packet}(@L,S,D,DT), \\ & & \quad \mathsf{route}(@L,D,N). \\ r2 & \mathsf{recv}(@L,S,D,DT) & :- \ \mathsf{packet}(@L,S,D,DT), D == L. \end{array}$$

Figure 1: An NDLog program for packet forwarding

To illustrate NDLog, we show an example query (Figure 1) that recursively forwards packets in a network. A typical NDLog program is composed of a set of rules. Each rule takes on the format $\mathsf{p} :- \mathsf{q}_1, \mathsf{q}_2, ...\mathsf{q}_n$, where p is a relation called the rule head, and q_is are rule bodies that are either relational atoms, arithmetic atoms or user-defined functions. Relations and rules of an NDLog program can be deployed in a distributed fashion. To logically specify the location of each relation, an "@" symbol – called the location specifier – is prepended to the first attribute of each relation.

Each node in the network maintains a database storing base tuples (i.e., tuples that are input by the user) and/or derived tuples (i.e., tuples that are generated by the NDLog program). During program execution, when all rule bodies of a rule r have corresponding tuples in the local database, r will be triggered, generating the head tuple. If the location specifier of the head tuple is different from that of bodies (e.g., *r1* in Figure 1), the head tuple will be transmitted through the network to the remote node. In the example program of Figure 1, *r1* forwards a local packet (packet) to neighbor N by looking up the packet's destination D in the local routing table (route). *r2* receives a packet and stores it locally in the recv table, if the packet is destined to the local node ($D == L$).

2.2 Distributed Network Provenance

Data provenance [7] can be used to explain why and how a given tuple is derived. Prior work [27] proposes network provenance, which faithfully records the execution of (possibly erroneous) applications in a (possibly misconfigured) distributed system. This allows the network administrators to inspect the derivation history of system states. For example, suppose there is a direct link between *n1* and *n3* in Figure 2. If the user prefers the routing with the shortest paths, the routing entry of *n1* in Figure 2 would have been erroneous – a correct entry should be $\mathsf{route}(@n1, n3, n3)$. The provenance engine, agnostic of this error, would record the packet traversal on the path $n1 \rightarrow n2 \rightarrow n3$. The user can later use the recorded provenance as an explanation on why the packet took a particular route, eventually leading to further investigation into the route table at *n1*.

Network provenance is typically represented as a directed tree rooted at the queried tuple. Figure 3 shows the provenance tree of a tuple $\mathsf{recv}(@n3, n1, n3, \text{"data"})$ derived from

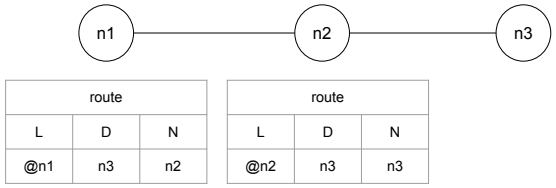

Figure 2: An example deployment of packet forwarding. Node *n1* and node *n2* has a local *route* table indicating routes towards node *n3*.

a packet $\mathsf{packet}(@n1, n1, n3, \text{"data"})$. The provenance tree is generated as $\mathsf{packet}(@n1, n1, n3, \text{"data"})$ traverses the network from node *n1* to *n3* in Figure 2. There are two types of nodes in a typical provenance tree: the rule nodes and the tuple nodes. The rule nodes (i.e., the oval nodes in Figure 3) stand for the rules that are triggered in the program execution, while the tuple nodes (i.e., the square nodes in Figure 3) represent tuples that trigger/are derived by the rule execution. Note that the root of a provenance tree is always a tuple node that represents the queried tuple.

To maintain the provenance, traditional database work [9] often stores data provenance along with the target tuple for efficient provenance querying. Such centralized provenance storage turns out to be very costly for the provenance in a network setting, which is typically constructed in a distributed fashion. In some cases, given the distributed nature of the application, it may also not be feasible to collect the information in a centralized fashion.

ExSPAN [27], a representative distributed provenance engine, maintains the provenance information in a distributed relational database. There are two (distributed) tables in the database: the *prov* table and the *ruleExec* table. The *prov* table records the provenance information for the direct derivation of a given tuple, while the *ruleExec* table maintains the information of a specific rule instance, including the rule name and the body tuples used in the rule evaluation. Table 1 shows an example relational database storing the provenance tree in Figure 3. Both tables are partitioned and maintained in a distributed fashion, according to the values of **Loc** and **RLoc** in each tuple.

ExSPAN uses a recursive provenance query to retrieve the provenance tree of a queried tuple. For example, to query the provenance tree for $\mathsf{recv}(@n3, n1, n3, \text{"}data\text{"})$ (Figure 3), ExSPAN first computes the hash value *vid6* of the tuple, and uses *vid6* to find the tuple $\mathsf{prov}(n3, vid6, rid3, n3)$ in the prov table. ExSPAN further uses the values *rid3* and *n3* in the tuple to locate $\mathsf{ruleExec}(n3, rid3, r2, (vid5))$ in the ruleExec table, which represents the provenance node for the rule execution (i.e., *r2*) that derives *vid6*. To further query the provenance of the body tuples that trigger *r2*, the querier would then look up (*vid5*) in the prov table. This recursive query processing continues until it reaches the base tuples (e.g., $\mathsf{route}(@n1, n3, n2)$).

We adopt the relational database storage model of ExSPAN. However, our provenance compression scheme applies generally to any distributed provenance model.

2.3 Motivation for Provenance Compression

A key problem not addressed in prior work on network provenance [27][26] is that the provenance information can become very large, especially for distributed applications

prov			
Loc	**VID**	**RID**	**RLoc**
n3	**vid6** (sha1(recv(@$n3, n1, n3,$ "$data$")))	**rid3**	n3
n3	**vid5** (sha1(packet(@$n3, n1, n3,$ "$data$")))	**rid2**	n2
n2	**vid4** (sha1(packet(@$n3, n1, n3,$ "$data$")))	**rid1**	n1
n2	**vid3** (sha1(route(@$n2, n3, n3$)))	NULL	NULL
n1	**vid2** (sha1(packet(@$n1, n1, n3,$ "$data$")))	NULL	NULL
n1	**vid1** (sha1(route(@$n1, n3, n2$)))	NULL	NULL

ruleExec			
RLoc	**RID**	**R**	**VIDS**
n3	**rid3**(sha1($r2$+n3+**vid5**))	$r2$	(**vid5**)
n2	**rid2**(sha1($r1$+n2+**vid3**+**vid4**))	$r1$	(**vid3**,**vid4**)
n1	**rid1**(sha1($r1$+n1+**vid1**+**vid2**))	$r1$	(**vid1**,**vid2**)

Table 1: Relational tables (ruleExec and prov) maintaining the provenance tree in Figure 3.

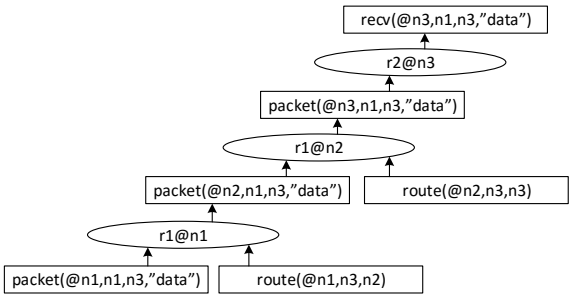

Figure 3: A (distributed) provenance tree for the execution of packet(@$n1, n1, n3,$ "$data$"), which traverses from node *n1* to node *n3* in Figure 2.

(e.g., network protocols) where event tuples trigger rules in a streaming fashion. For example, in Figure 2, if *n1* initiates a large volume of traffic towards *n3*, each packet in the traffic would generate a provenance tree similar to the one in Figure 3. Given that today's routers forward packets at rates over millions of packets per second, this would incur prohibitively high storage overhead for distributed provenance maintenance on each intermediate node.

We observe however that the provenance of different packets share significant similarities in their structures, presenting opportunities for provenance compression across different provenance trees. For example, in Figure 2, whenever a new packet is sent from *n1* to *n3*, an entire provenance tree is created and maintained. However, it is not hard to observe that all the packets traversing through *n1* and *n2* take the same route – that is, they join with the same local route tuples. Therefore, the storage for the provenance trees generated by these packets could be significantly reduced if we manage to remove the observed redundancy.

The key challenge of provenance compression in a distributed system is to achieve significant storage savings while incurring low network overhead (e.g., extra bandwidth and computation), and still enabling the user to query the provenance information effectively as does uncompressed provenance. Hence, we avoid content-level compression techniques such as gzip, but opt for the conservative compression based on the structure of provenance trees.

3. MODEL

A distributed system DS is modeled as an undirected graph $G = (V, E)$. Each node N_i in V represents an entity in DS. Two nodes N_i and N_j can communicate with each other if and only if there is an edge (N_i, N_j) in E. In DS, each node N_i maintains a local state in the form of a relational database DB_i. Tables in DB_i can be divided into *base tables* and *derived tables*. Tuples in *base tables* are manually updated, while tuples in *derived tables* are derived by network applications. Figure 2 is an example distributed system with three nodes.

3.1 Network applications

Each node in DS runs a number of network applications, which are specified in NDLog with syntactic restriction. The syntactic restriction enables efficient provenance compression (Section 5), while still being expressive enough to model most network applications. In particular, we have:

Definition 1. *An NDLog program Prog=*$\{r_1, r_2, ..., r_n\}$ *is a* distributed event-driven linear program *(DELP), if Prog satisfies the following three conditions:*

- Each rule is event-driven. *Each rule* r_i *can be specified in the form:* $[head] :- [event], [conditions]$*, where* $[event]$ *is a body relation designated by the programmer, and* $[conditions]$ *are all non-event body atoms.*
- Consecutive rules are dependent. *For each rule pair* (r_i, r_{i+1}) *in Prog, the head relation hd of* r_i *is identical to the event relation ev in* r_{i+1}*.*
- Head relations can only be event relations. *For each head relation hd in any rule* r_i*, there does not exist a rule* r_j*, such that hd is a non-event relation in* r_j*.*

In a typical network application, non-event relations often represent the network states, which change slowly compared to the fast rate of incoming events. For example, in the packet forwarding program, the route relation is either updated manually or through a network routing protocol. In either case, it changes slowly compared to the large volume of incoming packets. Therefore, we call the non-event relations in a DELP as *slow-changing* relations, and assume that they do not change during the fixpoint computation. This assumption is realistic and can be enforced easily in the networks where configurations are updated at runtime and packets see only either the old or new configuration version across routers, as shown in prior work [18] in the networking community.

A DELP $\{r_1, r_2, ..., r_n\}$ can be deployed in a distributed fashion over a network, and its execution follows the pipelined semi-naïve evaluation strategy introduced in prior work [10] – whenever an event tuple arrives at a node N_i, it triggers r_1 by joining with the slow-changing tuples at N_i. The generated head tuple hd is then sent to the node N_j – N_j is identified by the location specifier in hd – triggering r_2 at N_j. This process continues until r_n is executed.

DELP can model a large number of network applications, due to their event-driven nature, such as packet forwarding (Figure 1), Domain Name Service (DNS) resolution [12], Dynamic Host Configuration Protocol (DHCP) [6] and Address Resolution Protocol (ARP) [17].

3.2 Provenance for Network Applications

It is often the case that network administrators will use a subset of network states as their starting point for debugging. For example, in the packet forwarding program,

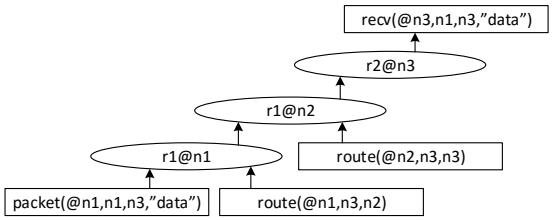

Figure 4: An optimized provenance tree for the tree in Figure 3.

prov			
Loc	**VID**	**RID**	**RLoc**
n3	**vid6**	**rid3**	n3

ruleExec					
RLoc	**RID**	**R**	**VIDS**	**NLoc**	**NRID**
n3	**rid3**	$r2$	NULL	n2	**rid2**
n2	**rid2**	$r1$	(**vid4**)	n1	**rid1**
n1	**rid1**	$r1$	(**vid1**,**vid2**)	NULL	NULL

Table 2: Optimized ruleExec and prov tables for the provenance tree in Figure 4

if a packet arrives at an unexpected destination node, the administrator may initiate a provenance query on the provenance of each recv tuple, but care less about the provenance of the packet tuples that traverse intermediate nodes. To satisfy this need, we allow the user to specify the *relations of interest* – i.e., relations whose provenance information interests the user the most in a network application – and our runtime system only maintains the *concrete and complete* provenance information for those tuples in the relations of interest. For the relations of less interest to the user, we can adopt the reactive maintenance strategy proposed by DTaP [26], by only maintaining those non-deterministic input tuples, and replaying the whole system execution to construct the provenance information during querying.

As with network provenance in prior works [27], we represent the provenance information of the tuples of interest as a tree. However, given the syntactic restriction we have for DELP programs, the provenance trees in our system do not maintain sub-provenance trees for the slow-changing tuples, such as the route tuples in Figure 3, even through these tuples may be derived from another network application, e.g., a routing protocol. To obtain the provenance tree of, say, a derived route tuple, the user could specify the route relation as the relation of interest in the application that derives it.

In the rest of paper, we use provenance trees to refer to the distributed provenance trees for DELP programs.

4. BASIC STORAGE OPTIMIZATION

Based on the model introduced in the previous section, we propose our basic storage optimization for provenance trees, which lays the foundation for the compression scheme in Section 5. For each provenance tree *prov*, we remove the provenance nodes representing the intermediate tuples that do not belong to the relations of interest. For example, in the packet forwarding program, assume that the user only specifies *recv* as the relation of interest, then the provenance tree *tr* in Figure 3 would be optimized into the tree tr' in Figure 4. The (distributed) relational database maintaining tr' is shown in Table 2, where *vid* values and *rid* values are identical to those in Table 1.

Compared to Table 1, Table 2 differs at two parts:

- The prov table only maintains the provenance for the queried tuple, i.e., the *recv* tuple. Other entries in the prov table are omitted because they represent either the removed intermediate tuples or the base tuples.
- Two extra columns **NLoc** and **NRID** are added to the *ruleExec* table. They help the recursive query find the child node for each provenance node in the tree.

The optimization of removing the intermediate nodes saves a fair amount of storage space, especially when the input events arrive at a high rate and generate a large number of intermediate tuples, as is common in typical networking scenarios. We use the query of recv($@n3, n1, n3,$ "*data*") in Table 2 to illustrate the two-step provenance querying process after the optimization:

Step 1: Construct the optimized provenance tree. The query first fetches the optimized provenance tree in a similar way to ExSPAN. Starting from the prov entry corresponding to recv($@n3, n1, n3,$ "*data*"), we fetch the provenance node for the last rule execution *rid3* in the ruleExec table, then follow the values in **NLoc** and **NRID** to recursively fetch all the ruleExec tuples (i.e., *rid3*, *rid2* and *rid1*) until no further provenance nodes can be fetched: both **NLoc** and **NRID** have *NULL* as their values.

Step 2: Compute the intermediate provenance nodes. At the end of Step 1, we obtain the provenance tree tr' in Figure 4. To construct the intermediate provenance nodes, we start from the leaf nodes, i.e., packet($@n1, n1, n3,$ "*data*") and route($@n1, n3, n2$), and re-execute the rule *r1* to derive packet($@n2, n1, n3,$ "*data*"). This process is repeated in a bottom-up fashion to construct all the intermediate tuples in Figure 3 until the root is reached.

The basic optimization still allows the user to query the complete provenance trees, but incurs extra computational overhead during the provenance querying to recover the intermediate nodes. The extra query latency is negligible, as is shown in Section 6.1.3.

5. EQUIVALENCE-BASED COMPRESSION

The storage optimization described in Section 4 focuses on reducing the storage overhead *within* a single provenance tree. Building upon this optimization, we further explore removing redundancy *across* provenance trees. We propose grouping provenance trees of a DELP program into equivalence classes, and only maintaining one copy of the shared sub-tree within each equivalence class. Our definition of the equivalence relation allows equivalent provenance trees to be quickly identified through the inspection of equivalence keys – a subset of attributes of the input event tuples – and compressed efficiently at runtime. The equivalence keys can be obtained through static analysis of the DELP.

5.1 Equivalence Relation

We first introduce the equivalence relation for provenance trees. We say that two provenance trees *tr* and tr' are equivalent, written ($tr \sim tr'$) if (1) they are structurally identical – i.e., they share the identical sequence of rules – and (2) the slow-changing tuples used in each rule are identical as well. Specifically, two equivalent trees *tr* and tr' only differ

at two nodes: (1) the root node that represents the output tuple and (2) the input event tuple. The formal definition of $tr \sim tr'$ can be found in Appendix C.2. In our packet forwarding example, the provenance tree generated by a new event packet$(@n1, n1, n3,$ "url") (with "url" as its payload) is equivalent to the tree in Figure 4.

For each equivalence class, we only need to maintain one copy for the sub-provenance tree shared by all the class members, while each individual member in the equivalence class only needs to maintain a small amount of delta information – i.e., the root node, the event leaf node, and a reference to the shared sub-provenance tree. Additionally, this definition of the equivalence relation has the benefit of identifying equivalent provenance trees more efficiently than traditional node-by-node comparison. In fact, we show that the equivalence of two provenance trees can be determined by the equivalence of the input event tuples in both trees, based on the observation that the execution of a DELP is uniquely determined by the values of a subset of attributes in the input event tuple. For example, in the packet forwarding program (Figure 1), if the values of the attributes (loc, dst) in two input packet tuples are identical, these two tuples will generate equivalent provenance trees.

We denote the minimal set of attributes K in the input event relation whose values determine the provenance trees as *equivalence keys*. Two event tuples ev_1 and ev_2 of a relation e are said to be *equivalent w.r.t* K, written $ev_1 \sim_K ev_2$, if their valuation of K is equal. Formally:

Definition 2 (Event equivalence)**.** *Let* $K = \{e{:}i_1, \cdots, e{:}i_m\}$, $e(t_1 \cdots t_n) \sim_K e(s_1 \cdots s_n)$ *iff* $\forall j \in \{i_1, \cdots, i_m\}$, $t_j = s_j$.

Here, $e{:}i$ denotes the i^{th} attribute of the relation e.

Based on the above discussion, our approach to compressing provenance trees, with regard to a program DQ, consists of the following two main algorithms. (1) an equivalence keys identification algorithm, which performs static analysis of DQ to compute the equivalence keys (Section 5.2); and (2) an online provenance compression algorithm, which maintains the shared provenance tree for each equivalent class in a distributed fashion (Section 5.3).

The correctness of using the event equivalence for determining the provenance tree equivalence is shown in Theorem 1. The proof is discussed in Section 5.2.

Theorem 1 (Correctness of equivalence keys)**.** *Given a program DQ of DELP, and two input event tuples* ev_1 *and* ev_2*, if* $ev_1 \sim_K ev_2$*, where K are the equivalence keys for DQ, then for any provenance tree* tr_1 *(*tr_2'*) generated by* ev_1 *(*ev_2*), there exists a provenance tree* tr_2 *(*tr_1'*) generated by* ev_2 *(*ev_1*) s.t.* $tr_1 \sim tr_2$ *(*$tr_1' \sim tr_2'$*).*

5.2 Equivalence Keys Identification

Given a DELP, we define a static analysis algorithm to identify the equivalence keys of the input event relation. The algorithm consists of two steps: (1) building an attribute-level dependency graph reflecting the relationship between the attributes of different relations and (2) computing equivalence keys based on the constructed dependency graph. Details of each step are given below.

Build the attribute-level dependency graph. An attribute-level dependency graph $G{=}(V,\ E)$ is an undirected graph. Nodes of G represent the attributes in relations. Specifically, for the i-th attribute of a relation *rel*, a vertex *vtx* is created in G, labeled as $(rel{:}i)$. We refer interested readers to Appendix F for an example graph of the packet forwarding program.

```
1: function GetEquiKeys(G, ev)
2:     eqid ← {}
3:     eqid.append(ev:0)
4:     nodes ← event attribute nodes in G
5:     for each ev:i in nodes do
6:         for bnode in non-event nodes of G do
7:             if ev:i is reachable to bnode then
8:                 eqid.append(ev:i)
9:     return eqid
10: end function
```

Figure 5: Pseudocode to identify equivalence keys

Two vertices *v1* and *v2* are connected in G if and only if *v1* represents an attribute $attr_1$ of the event relation in a rule r and *v2* represents another attribute $attr_2$ in r, and satisfies any of the following conditions: (1) $attr_2$ is an attribute with the same name as $attr_1$ in a slow-changing relation (e.g. *v1* = packet:1 and *v2* = route:1 in rule $r1$ of Figure 1); (2) $attr_2$ is a head attribute with the same name as $attr_1$ (e.g., *v1* = packet:1 and *v2* = recv:1 in $r2$ of Figure 1); (3) $attr_2$ is an attribute with the same name as $attr_1$ in an arithmetic atom (e.g. *v1* = (packet:0) and *v2* = $((D == L).left{:}0)$ in rule $r2$ of Figure 1); and (4) *v1* is on the right hand side of an assignment *asn* and $attr_2$ is on the left hand side of *asn*. (e.g., if rule $r2$ of Figure 1 were to be redefined as $r2'$ recv$(@L, S, N, DT) :-$ packet$(@L, S, D, DT), N := L{+}2.$, and *v1* = (packet:0) while *v2* = (recv:2)).

Identify equivalence keys. Given the attribute-level dependency graph G, we identify the equivalence keys of the event relation *ev* using the function GetEquiKeys (Figure 5). GetEquiKeys takes G and *ev* as input, and outputs a list of attributes *eqid* representing the equivalence keys. In the algorithm, for each node $(ev{:}i)$ in G, GetEquiKeys checks whether $(ev{:}i)$ is reachable to any node corresponding to an attribute in a slow-changing relation, an arithmetic atom, or an assignment. If this is the case, $(ev{:}i)$ would be identified as a member of the equivalence keys, and appended to *eqid*. We always include the attribute indicating the input location of *ev* (e.g., (packet:0)) in the equivalence keys, to ensure that the input event tuples on different network nodes have different equivalence keys. When applied to the packet forwarding program, GetEquiKeys would identify (packet:0) and (packet:2) as equivalence keys.

To prove Theorem 1, we introduce the following denotations. We use predicate joinSAttr$(p{:}n)$ to denote that a node $p{:}n$ in the dependency graph has an edge to an attribute in a slow changing relation, an arithmetic atom or an assignment. We denote each edge between two attributes $(p{:}n, q{:}m)$ of tables that are not slow-changing (i.e., event tuple and intermediary tuples) as predicate joinFAttr$(p{:}n, q{:}m)$. We inductively define connected$(e{:}i, p{:}n)$ to denote a path in the graph from the node $(e{:}i)$ to the node $(p{:}n)$ (using joinFAttr$(p{:}n, q{:}m)$ predicates). We then formally define what it means for K to be equivalence keys, given a DELP as follows:

Definition 3. *K are the equivalence keys for a program DQ of DELP, if* $\forall e{:}i \in K$*, either* $DQ \vdash$ joinSAttr$(e{:}i)$ *or* $\exists p, n$ *s.t.* $DQ \vdash$ connected$(e{:}i, p{:}n)$ *and* $DQ \vdash$ joinSAttr$(p{:}n)$.

Instead of directly proving Theorem 1, we prove a stronger lemma below that gives us Theorem 1 as a corollary. In Lemma 2, we write $tr : P$ to mean that tr is a derivation tree of the output tuple P, and write $DQ, \mathcal{DB}, ev \vDash tr : P$ to mean that tr is a derivation tree for P using the program DQ, a database of materialized tuples $\mathcal{DB}$, and the event tuple ev.

Lemma 2 (Correctness of equivalence keys (Strong))**.**
If GetEquiKeys$(G, ev) = K$ *and* $ev_1 \sim_K ev_2$
and $DQ, \mathcal{DB}, ev_1 \vDash tr_1 : p(t_1, ..., t_n)$,
then $\exists tr_2 : p(s_1, ..., s_n)$ *s.t.* $DQ, \mathcal{DB}, ev_2 \vDash tr_2 : p(s_1, ..., s_n)$
and $tr_1 : p(t_1, ..., t_n) \sim tr_2 : p(s_1, ..., s_n)$
and $\forall i \in [1, n]$, $t_i \neq s_i$ *implies*
$\exists \ell$ *s.t.* $DQ \vdash$ connected$(e{:}\ell, p{:}i)$ *and* $\ell \notin K$.

Intuitively, Lemma 2 states that given two equivalent input event tuples ev_1 and ev_2 w.r.t. K, and ev_1 generates a provenance tree tr_1, we can construct a tr_2 for ev_2 such that tr_1 and tr_2 are equivalent – i.e., they share the same structure and slow changing tuples. Furthermore, if the two output tuples $p(t_1, ..., t_n)$ and $p(s_1, ..., s_n)$ have different values for a given attribute, this attribute must connect to a non-equivalence keys attribute in the dependency graph. This last condition allows for an inductive proof (Appendix C.3.2) of Lemma 2 over the structure of the tree.

Time complexity. Next, we analyze the time complexity of static analysis. Assume the DELP program DQ has m rules. Each rule r has k atoms, including the head relation and all body atoms. Each atom has at most t attributes. Hence, the attribute-level dependency graph G has at most $n{=}m * k * t$ nodes. The construction of G takes $O(n^2)$ time, and the identification of equivalence keys takes $O(t * n)$ time. Normally t is much smaller than n. Therefore, the total complexity of static analysis is $O(n^2)$.

5.3 Online Provenance Compression

We next present an online provenance compression scheme that compresses equivalent (distributed) provenance trees based on the identified equivalence keys. In our compression scheme, each execution of a DELP program DQ, triggered by an event tuple ev, is composed of three stages:

- **Stage 1: Equivalence keys checking.** Extract ev's equivalence keys values v, and check whether v has ever been seen from previous event tuples. If so, set a Boolean flag *existFlag* to *True*. Otherwise, set *existFlag* to *False*. Then tag *existFlag* along with ev.
- **Stage 2: Distributed online provenance maintenance.** If *existFlag* is *True*, no provenance information is maintained for each rule execution. Otherwise, the provenance nodes for the rule execution are maintained in a distributed fashion.
- **Stage 3: Output tuple provenance maintenance.** When the execution finishes, associate the output tuple to the shared provenance tree, to allow for future provenance querying.

To illustrate this, Figure 6 presents an example consisting of two packets traversing the network topology (from n1 to n3) in Figure 2. packet$(@n1, n1, n3,$ "data") is first inserted for execution (represented by the solid arrows), followed by the execution of packet$(@n1, n1, n3,$ "url") (represented by the dashed arrows). The three stages of online compression are logically separated with vertical dashed lines. Table 3 presents the (distributed) relational tables (i.e., the ruleExec table and the prov table) that maintain the compressed provenance trees for both executions. Next, we introduce each stage in detail.

Equivalence Keys Checking. Upon receiving an input event ev, our runtime system first checks whether the values of ev's equivalence keys have been seen before. To do this, we use a hash table *htequi* to store all unique equivalence keys that have arrived. If ev's equivalence keys *eqid* already exists in *htequi*, a Boolean flag *existFlag* will be created and set to *True*. This *existFlag* is supposed to accompany ev throughout the execution, notifying all nodes involved in the execution to avoid maintaining the concrete provenance tree. Otherwise, if *eqid* does not exist in *htequi*, *existFlag* would be set to *False*, notifying the subsequent nodes that a provenance tree should be concretely maintained. For example, in Figure 6, when the first packet tuple $packet(@n1, n1, n3,$ "data") arrives, it has values $(n1, n3)$ for its equivalence keys, which have never been encountered before, so its *existFlag* is *False*. But when the second packet tuple $packet(@n1, n1, n3,$ "url") arrives, since it shares the same equivalence keys values with the first packet, the *existFlag* for it is *True*.

Distributed Online Provenance Maintenance. For each rule r triggered in the execution, we selectively maintain the provenance information based on *existFlag*'s value. if *existFlag* is *False*, the provenance nodes are maintained as tuples in the *ruleExec* table locally. Otherwise, no provenance information is maintained at all. For example, in Figure 6, when packet$(@n2, n1, n3,$ "data") triggers rule *r1* at node *n2*, the *existFlag* is *False*. Therefore, we insert a tuple $ruleExec(n2, rid2, r1, vid1, n1, rid3)$ into the *ruleExec* table at node n2 to record the provenance. The semantics of the inserted tuple are the same as introduced in Section 4. In comparison, when packet$(@n2, n1, n3,$ "url") triggers *r2* at node *n2*, its *existFlag* is *True*. In this case, we simply execute *r2* without recording any provenance information.

Output Tuple Provenance Maintenance. For the execution whose *existFlag* is *True*, we need to associate its output tuple to the shared provenance tree maintained by the execution whose *existFlag* is *False*. To do this, we use a hash table *hmap* to store the reference to the shared provenance tree. The key of *hmap* is the hash value of the equivalence keys, and the value is the node closest to the root in the shared provenance tree. For example, in Figure 6, the provenance tree shared by two executions are stored in *hmap* as $\{hash(n1, n3)$: $(n3, rid1)\}$.

We then associate an output tuple tp to the shared provenance tree st, by looking up its equivalence keys' values in *hmap*. The association is stored as a tuple in the prov table. For example, in Figure 6, the first execution generates the output tuple recv$(@n3, n1, n3,$ "$data$"), which is associated to the node closest to the root of the shared provenance tree $((n3, rid1))$. This association is reflected by the tuple prov$(n3, tid1, n3, rid1, evid1)$ in the prov table (Table 3). *evid1* in the prov tuple is used to identify the event tuple peculiar to the execution, which is not included in the shared provenance tree.

Correctness of Online Compression. We prove the correctness of the online compression algorithm by showing that the distributed provenance elements maintained in the ruleExec and prov tables contain the exact same set of provenance trees of tuples derived by a semi-naïve evaluation (Theorem 3). We define the operational semantics

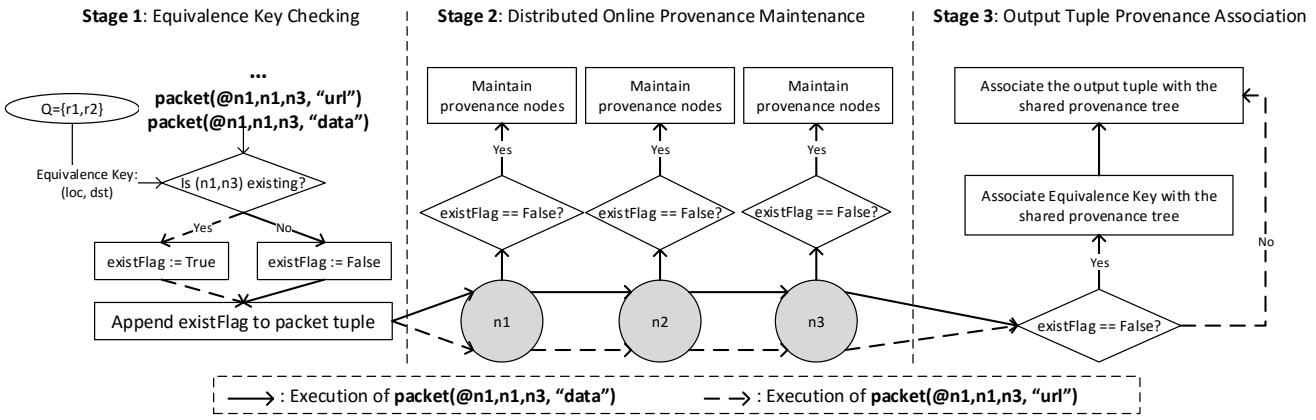

Figure 6: An example execution of the packet forwarding program in Figure 1. The program is first triggered by packet($@n1, n1, n3,$ "data"), followed by packet($@n1, n1, n3,$ "url").

ruleExec					
Loc	**RID**	**RULE**	**VIDS**	**NLoc**	**NRID**
n3	**rid1**(sha1(r2))	r2	NULL	n2	**rid2**
n2	**rid2**(sha1(r1,*vid1*))	r1	(*vid1*(sha1(route($@n2, n3, n3$))))	n1	**rid3**
n1	**rid3**(sha1(r1,*vid2*))	r1	(*vid2*(sha1(route($@n1, n3, n2$))))	NULL	NULL

prov				
Loc	**VID**	**RLoc**	**RID**	**EVID**
n3	**tid1**(sha1(recv($@n3, n1, n3,$ "data")))	n3	**rid1**	**evid1**(sha1(packet($@n1, n1, n3,$ "data")))
n3	**tid2**(sha1(recv($@n3, n1, n3,$ "url")))	n3	**rid1**	**evid2**(sha1(packet($@n1, n1, n3,$ "url")))

Table 3: ruleExec table and prov table for compressed provenance trees produced by Figure 6

of the semi-naïve evaluation of the program using a set of transition rules of form: $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$, where $\mathcal{C}_{sn}$ denotes the state of the semi-naïve evaluation that stores the full derivation trees as provenance [5]. We also define a set of transition rules of form: $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$ for the semi-naïve evaluation with our online compression algorithm. Here, $\mathcal{C}_{cm}$ denotes the state of the semi-naïve evaluation with compression. We can assemble entries in the ruleExec and prov tables to reconstruct a provenance tree. Given a provenance tree $\mathcal{P}$, we can also find the corresponding entries in the ruleExec and prov tables. This correspondence is denoted as $tr \sim_d \mathcal{P}$ and can be defined by induction over the structure of the provenance tree.

Theorem 3 (Correctness of Compression)**.** *$\forall n \in \mathbb{N}$ and initial state $\mathcal{C}_{init}$, $\mathcal{C}_{init} \rightarrow_{SN}^{n} \mathcal{C}_{sn}$ then exists $\mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \nearrow_{CM}^{n} \mathcal{C}_{cm}$ and for any derivation tree $tr \in \mathcal{C}_{sn}$, there exists a provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$ s.t. $tr \sim_d \mathcal{P}$ and for all provenance trees $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a derivation tree $tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d \mathcal{P}$. And the same is true for the semi-naïve when $\mathcal{C}_{init} \nearrow_{CM}^{n} \mathcal{C}_{cm}$.*

The above theorem states that if we execute a DELP DQ from an initial state $\mathcal{C}_{init}$ (no derivations are generated yet) in n steps to a state $\mathcal{C}_{sn}$, then we can execute the same program starting from the same initial state using the online compression scheme. In the end, the ending state has the same provenance. An implication of Theorem 3 is that the compressed provenance trees, like traditional network provenance, would faithfully record the system execution, even if the execution is erroneous due to misconfiguration (e.g., wrong routing tables).

Theorem 3 is a corollary of Lemma 4, which shows that the semi-naïve evaluation with the online compression scheme is bisimilar to the one that stores the full derivation trees. The bisimilarity relation shows that the provenance trees stored by both evaluations have the same semantics.

Lemma 4 (Compression Simulates Semi-naïve)**.** *$\forall n \in \mathbb{N}$ given initial state $\mathcal{C}_{init}$, and $\mathcal{C}_{init} \rightarrow_{SN}^{n} \mathcal{C}_{sn}$ then $\exists \mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \nearrow_{CM}^{n} \mathcal{C}_{cm}$ and $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$ and vice versa.*

We define a relation $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ between $\mathcal{C}_{sn}$ and $\mathcal{C}_{cm}$ such that $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ is a bisimulation relation: if $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$, then $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$ implies there exists a state $\mathcal{C}_{cm}'$ s.t. $\mathcal{C}_{cm} \rightarrow_{SN} \mathcal{C}_{cm}'$ and $\mathcal{C}_{sn}'$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}'$ and vice versa. The formal definition of $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$ is presented in Appendix G.1.1. Intuitively, $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ relates two configurations that have the same program, the same program execution state, and most importantly, any provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a derivation tree $tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d \mathcal{P}$ and vice versa. This definition is complex due to the distributed nature of the compression and the possibility that tuples arrive out of order.

Proof details are given in Appendix G.1. Briefly, we apply induction over n, the number of steps taken by the execution. The key is to show that if one configuration takes a step, the other configuration takes the same step and the resulting states are again bisimular.

Generality of equivalence-based compression. The idea of equivalence-based compression is not just applicable to distributed scenarios, but can be generally used to compress arbitrary provenance tree sets maintained in a centralized manner as well. We adopt the definition of the equivalence relation in Section 5.1 because it allows us to use

equivalence keys to efficiently identify equivalent provenance trees, thus more suitable for the distributed environment where networking resources (e.g., bandwidth) are scarce.

5.4 Inter-Equivalence Class Compression

The online compression scheme introduced in Section 5.3 focuses on intra-equivalence class compression of the provenance trees – i.e., only the trees of the same equivalence class are compressed. In fact, the provenance trees of different equivalence classes can be compressed as well. For example, assume a tuple packet($@n2, n2, n3,$ "ack") is inserted into *n2* in Figure 6 for execution. The produced provenance tree *prov* shares the provenance nodes *rid1* and *rid2* in the ruleExec table of Table 3. To avoid the storage of such redundant rule execution nodes, we separate the ruleExec table into two sub-tables: the ruleExecNode table and the ruleExecLink table. The ruleExecNode table maintains the concrete rule execution nodes, while the ruleExecLink table, which is maintained for each provenance tree *tr* individually, records the parent-child relationship of the rule execution nodes in *tr*. Table 4 presents the ruleExecNode table and the ruleExecLink table for the ruleExec table in Table 3. If two provenance trees, whether in the same equivalence class or not, share the same rule execution node *nd*, only one copy of the concrete *nd* will be maintained in the ruleExecNode table. Each tree maintains a reference pointer pointing to *nd* in their respective ruleExecLink tables.

ruleExecNode			
Loc	**RID**	**RULE**	**VIDS**
n3	**rid3**	r2	NULL
n2	**rid2**	r1	(**vid1**)
n1	**rid1**	r1	(**vid2**)

ruleExecLink			
Loc	**RID**	**NLoc**	**NRID**
n3	**rid3**	n2	**rid2**
n2	**rid2**	n1	**rid1**
n1	**rid1**	NULL	NULL

Table 4: The ruleExecNode table and the ruleExecLink table replacing the ruleExec table in Table 3 to allow for compression of the shared rule execution nodes

5.5 Updates to Slow-changing Tables

Though we assume that slow-changing tables do not change during a fixpoint computation, our system is designed to handle these updates at runtime. Figure 7 presents an example scenario based on Figure 2, where the network administrator decides to use *n4*, instead of *n2*, as the next hop for the packets sent from *n1* to *n3*. To redirect the traffic, the administrator (1) deletes the route entry route($@n1, n3, n2$), and (2) inserts a new route entry route($@n1, n3, n4$).

Deletion of a tuple from a slow-changing table, such as route($@n1, n3, n2$) in Figure 7, does not affect the stored provenance, as provenance information is monotone – that is, it represents the execution history which is immutable [26], thus independent of the change of slow-changing tables.

However, when a tuple *tp* is inserted into a slow-changing table, such as route($@n1, n4, n3$) in Figure 7, the provenance tree generated by *tp* could be incorrect or missing. For example, in Figure 7, after route($@n1, n4, n3$) is inserted, the provenance trees for all subsequent packets need to be re-

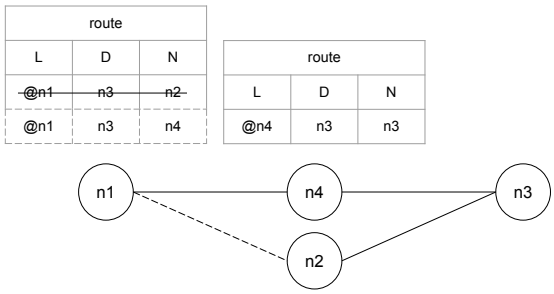

Figure 7: An updated topology of Figure 2. A new node *n4* is deployed to reach *n3*. The route table of *n1* is updated to forward packets to *n4* now.

calculated. However, since these packets are not the first in their equivalence classes, their *existFlag*s are set to *false*. As a result, the provenance tree for the packet traversal on the path $n1 \rightarrow n4 \rightarrow n3$ would not be maintained.

To handle such scenarios, we require that, once a new tuple *tp* is inserted into a node *n*'s slow-changing tables, *n* should broadcast a control message *sig* to all the nodes in the system. Any node receiving *sig* would reset the hash table used for equivalence keys checking (Section 5.3). Therefore, provenance trees will be maintained again for all equivalence classes. In Figure 7, after the insertion of route($@n1, n3, n4$), *n1* would broadcast a *sig* to all the nodes, including itself. When the next packet *pkt* destined to *n3* arrives at *n1*, the packet would have its *existFlag* set as *false*. When this packet traverses the path $n1 \rightarrow n4 \rightarrow n3$, the nodes on the path are expected to maintain the corresponding provenance nodes. In all our network applications, the extra network overhead incurred by the broadcast and the impact on the effectiveness of compression due to reset of the hash table is negligible, as slow-changing tables are updated infrequently in practice (relative to the rate of event arrival). We experimentally validated this in Section 6.

5.6 Provenance Querying

To query the provenance tree of an output tuple *tp*, we take the following steps:

- Compute the hash value *htp* of *tp*, and find the tuple *prvtp* in the prov table that has *htp* as the value of the **VID** attribute.
- Initiate a recursive query for the (shared) provenance nodes in the ruleExec table, starting with the values of (**Loc**, **RID**) in *prvtp*. Also, tag the event ID *evid* stored in the attribute **EVID** along with the query.
- When the recursive query reaches (*NULL*, *NULL*) for the attributes (**NLoc**, **NRID**) in a ruleExec tuple, the tagged *evid* is used to retrieve the event tuple materialized at the first node of the execution.

For example, in Table 3, to query the provenance tree of recv($@n3, n1, n3,$ "data"), we first find prov($n3, tid1, n3, rid1, evid1$), and use the values ($n3, rid1$) to initiate the recursive query in the ruleExec table to fetch the provenance nodes *rid1*, *rid2* and *rid3*. *evid* is carried throughout the query, and is used to retrieve the event packet($@n1, n1, n3,$ "data") when the query stops at ruleExec($n1$, $rid3$, $r1$, $vid2$, *NULL*, *NULL*). The above steps return a collection of entries from the ruleExec and prov tables. We define a top-level algorithm QUERY that reconstructs the full provenance tree *tr* based on these entries. The pseudocode can be found in Figure 33

in Appendix H.1.2. QUERY takes as input the network state $\mathcal{C}_{cm}$ of the online compression scheme, an output tuple P, an event ID *evid*, and returns a set of provenance trees, each of which corresponds to one derivation of P using the input event tuple with ID *evid*. The above example has only one derivation for the output tuple, so we return a singleton set.

Correctness of Querying. From the correctness of the online compression algorithm (Theorem 3), we can prove that all the provenance trees generated by the semi-naïve evaluation can be queried and the query algorithm will return the correct provenance tree. One subtlety is that the compression algorithm may propagate updates out of order, causing ruleExec entries to be referred to in a provenance tree before being stored. We handle this subtlety by assuming all updates are processed before querying.

Theorem 5 (Correctness of the Query Algorithm)**.**
$\forall n \in \mathbb{N}$, *given initial state* $\mathcal{C}_{init}$ *s.t.* $\mathcal{C}_{init} \rightarrow^{n}_{CM} \mathcal{C}_{cm}$
and there are no more updates to be processed,
then $\exists \mathcal{C}_{sn}$ *s.t.* $\mathcal{C}_{init} \rightarrow^{n}_{SN} \mathcal{C}_{sn}$
and $\forall tr$:P *in the output provenance storage of* $\mathcal{C}_{sn}$
s.t. hash(EVENTOF(tr)) = *evid,*
$\exists \mathcal{M}$ *s.t.* QUERY($\mathcal{C}_{cm}, P, evid$) = $\mathcal{M}$ *and* $tr \in \mathcal{M}$
and $\forall tr' \in \mathcal{M} \backslash tr$, tr' *is a proof of* P *stored in* $\mathcal{C}_{sn}$
and hash(EVENTOF(tr')) = *evid.*

Details of the proof are in Appendix **??**. Briefly, by Theorem 3), there exists $\mathcal{C}_{sn}$ s.t. $\mathcal{C}_{init} \rightarrow^{n}_{SN} \mathcal{C}_{sn}$ and $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$. By $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$, we know that for any tr of tuple P in $\mathcal{C}_{sn}$, there exists a corresponding provenance tuple *prov* in $\mathcal{C}_{cm}$ that stores an association to the root of some provenance tree $\mathcal{P}$ for P, and that tr corresponds to $\mathcal{P}$ ($tr \sim_d \mathcal{P}$). We induct over the depth of $\mathcal{P}$ to show that given the root of $\mathcal{P}$, the recursive lookup will return $\mathcal{P}$. Now, it is straighforward to reconstruct tr from $\mathcal{P}$, as the return value of QUERY.

6. EVALUATION

We have implemented a prototype based on enhancement to the RapidNet [13] declarative networking engine. At compile time, we add a program rewrite step that rewrites each DELP program into a new program that supports online provenance maintenance and compression at runtime. We evaluate our prototype to understand the effectiveness of the online compression scheme. In all the experiments, we compare three techniques for maintaining distributed provenance. The first is ExSPAN [27], a typical network provenance engine . We maintain uncompressed provenance trees in the same way as ExSPAN. The second is the distributed provenance maintenance with basic storage optimization (Section 4). The third is the provenance maintenance using equivalence-based compression (Section 5). In the evaluation section, we refer the three techniques as *ExSPAN*, *Basic*, and *Advanced* respectively.

Workloads. Our experiments are carried out on two classic network applications: packet forwarding (Section 2) and DNS resolution. DNS resolution is an Internet service which translates human-readable domain names into IP addresses. Both applications are event-driven, and typically involve large volume of traffic during execution. The high-volume traffic incurs large storage overhead if we maintain provenance information for each packet/DNS request, which leaves potential opportunity for compression. The workloads are also sufficiently different to evaluate the generality of our approach. Packet forwarding involve larger messages along different paths in a graph, while DNS lookups involve smaller messages on a tree-like topology.

Testbed. In our experiment setup, we write the packet forwarding and DNS resolution applications in DELP, and use our enhanced RapidNet [13] engine to compile them into low-level (i.e., C++) execution codes.

The experiments for measuring storage and bandwidth are run on the ns-3 [14] network simulator, which is a discrete-event simulator that allows a user to evaluate network applications on a variety of network topologies. The simulation is run on a 32-core server with Intel Xeon 2.40 GHz CPUs. The server has 24G RAM, 400G disk space, and runs Ubuntu 12.04 as the operating system. We run multiple node instances on the same machine communicating over the ns-3 simulated network.

Performance Metrics. The performance metrics that we use in our experiments are: (1) the storage overhead, and (2) the network overhead (i.e., bandwidth consumption) for provenance maintenance, and (3) the query latency when different provenance maintenance techniques are adopted.

In our experiments, the relational provenance tables are maintained in memory. To measure the storage occupation, we use the boost library [19] to serialize C++ data structures into binary data. At the end of each experiment run, we serialize the per-node provenance tables (i.e., ruleExec table and prov table) into binary files, and measure the size of files to estimate the storage overhead.

6.1 Application #1: Packet Forwarding

Our first set of results is based on the packet forwarding program in Figure 1. The topology we used for packet forwarding is a 100-node transit-stub graph, randomly generated by the GT-ITM [24] topology generator. In particular, there are four transit nodes – i.e., nodes through which traffic can traverse – in the topology, each connecting to three stub domains, and each stub domain has eight stub nodes – i.e., nodes where traffic only originates or terminates. Transit-transit links have 50ms latency and 1Gbps bandwidth; transit-stub links have 10ms latency and 100Mbps bandwidth; stub-stub links have 2ms latency and 50Mbps bandwidth. The diameter of the topology is 12, and the average distance for all node pairs is 5.3. Each node in the topology runs one instance of the packet forwarding program.

In the experiment, we randomly selected a number of node pairs (s, d) – where s is the source and d is the destination– and sent packets from s to d while the provenance of each packet is maintained. To allow the packets to be correctly forwarded in the network, we pre-computed the shortest path p between s and d using a distributed routing protocol written as a declarative networking program[11]. The routes are stored in the route tables at each node in p.

6.1.1 Storage of Provenance Trees

Figure 8 shows the CDF (Cumulative Distribution Function) graph of storage growth for all the nodes in the 100-node topology. In the experiment, we randomly selected 100 pairs of nodes, and continuously sent packets within each pair at the rate of 100 packets/second. As packets are transmitted, their provenance information is incrementally created and stored at each node (and optionally compressed for Basic and Advanced). We calculated the average storage

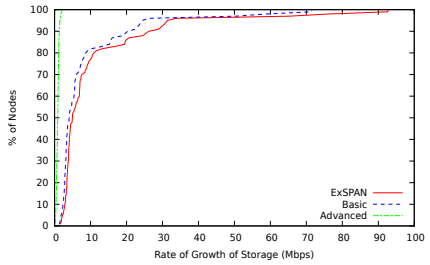

Figure 8: Cumulative growth rate of provenance with 100 pairs of communicating nodes, at input rate of 100 packets/second.

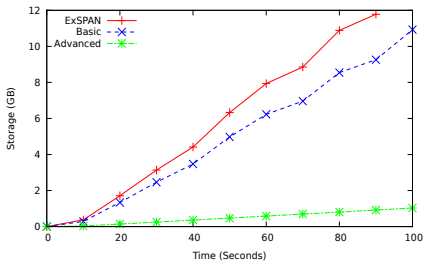

Figure 9: Provenance storage growth of all nodes, with input rate of 100 packets/second for 100 pairs of communicating nodes.

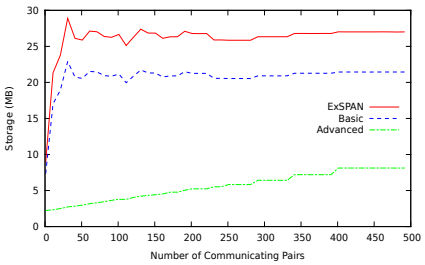

Figure 10: Provenance storage usage with 2000 input packets evenly distributed among increasing number of communicating pairs.

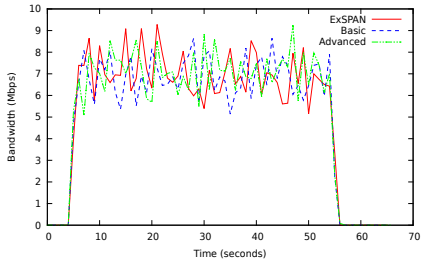

Figure 11: Bandwidth consumption during packet forwarding, with 500 pairs of nodes, each transmitting 100 packets.

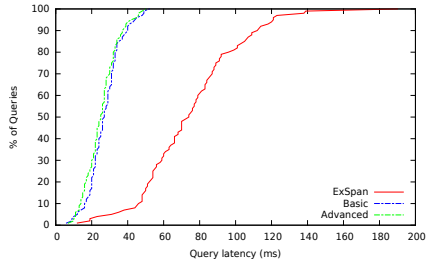

Figure 12: Cumulative distribution of provenance querying latency for 100 random queries with 100 pairs of communicating nodes.

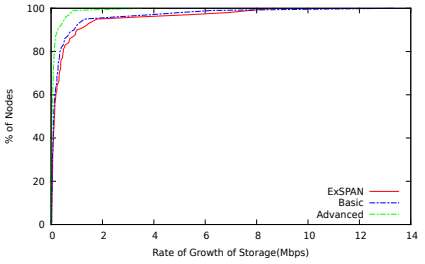

Figure 13: Cumulative provenance storage growth rate of nameservers with input request at a rate of 1000 requests/second.

growth rate of each node, and plotted a CDF graph based on the results. We observe that ExSPAN has the highest storage growth rate among the three: 20% of the nodes have storage growth greater than 5 Mbps; 4% of nodes (i.e., transit nodes) have storage growth greater than 30 Mbps. This is because a number of node pairs share the same transit node in their paths. As expected, Basic has less storage growth rate compared to ExSPAN, as it removes intermediate packet tuples from the provenance tables of each node. Advanced significantly outperforms the other two: all the nodes in the topology has less than 2 Mbps storage growth rate. The gap between Advanced and ExSPAN results from the fact that Advanced only maintains one representative provenance tree for each pair of nodes, while ExSPAN has to maintain provenance trees of all the traversing packets.

Figure 9 shows the total storage usage with continuous packet insertion. We ran the experiment for 100 seconds and took a snapshot of the storage every 10 seconds. The figure shows that ExSPAN has the highest storage overhead. For example, it reaches the storage of 11.8 GB at 90 seconds, and keeps growing in a linear fashion. Basic has a similar pattern, with 9.2 GB at 90 seconds. However, Advanced presents lower storage growth, where at 90 seconds it only consumes storage space of 0.92 GB. We further calculate the average growth rate for all three lines. ExSPAN's storage grows at 131 MB/second, Basic at 109 MB/second, and Advanced at 10.3 MB/second. This means that ExSPAN could fill a 1TB disk within 2 hours, Basic within 2.5 hours, whereas Advanced more than one day.

Figure 10 shows the storage usage when we increase the number of communicating pairs, but keep the total number of packets the same (i.e., 2000 packets). All the packets are evenly distributed among all the communicating pairs. We observe that the storage usage of ExSPAN and Basic remains almost constant: ExSPAN's total storage usage is around 27 MB and Basic's total storage usage is around 21 MB. This is because in both cases, each packet has a provenance tree maintained in the network, irrelevant of its source and destination. The burst of storage at the beginning of the experiments for ExSPAN and Basic is due to the fact that sizes of provenance trees also depend on the length of the path that each packet traverses. In our experiment, the initial node pairs happen to have path length shorter than the average path length in the topology, thus incurring less storage overhead.

For the case of Advanced, its storage usage increases with the number of communicating pairs. This is because each communicating pair forms an equivalence class, and maintains one copy of the shared provenance tree in the equivalence class. Therefore, whenever a new communicating pair is added to the experiment, we need to maintain one more provenance tree for that pair, which increases the total storage. Despite the storage increase, Advanced still consumes much less storage space than the other two schemes.

In summary, we observe that Basic is able to reduce storage growth, and in combination with the equivalence-based compression (Advanced), the storage reduction is significant – i.e., a 92% reduction over ExSPAN.

6.1.2 *Network Overhead.*

Figure 11 presents the bandwidth utilization when we randomly selected 500 pairs of nodes and each pair communicated 100 packets. As expected, the bandwidth consumption of Advanced is close to the ones of ExSPAN and Basic. This is because the extra information carried with each packets is merely *existFlag* and some auxiliary data (e.g., hash value of the event tuple), which is negligible compared to the large payload of the packets. We repeated the experiment

for Advanced, but updated a route every 10 seconds, in order to study the effects of updates to slow-changing tuples. We observe a negligable bandwidth increase of 0.6%.

6.1.3 Query Latency

To evaluate latency of queries, we used an actual distributed implementation that can account for both network delays and computation time. We ran the packet forwarding application on a testbed consisting of 25 machines. Each machine is equipped with eight Intel Xeon 2.67 GHz CPUs, 4G RAM and 500G disk space, running CentOS 6.8 as the operating system.

On each machine, we ran up to four instances of the same packet forwarding application with provenance enabled. Instead of communicating via the ns-3 network, actual sockets were used over a physical network. In total, there were 100 nodes, connected together using the same transit-stub topology we used for simulation.

In our experiment, we executed 100 queries, selected on random nodes, where each query returns the provenance tree for a recv tuple corresponding to a random source and destination pair, where the destination node is the starting point of the query. The query is executed in a distributed fashion as described in Section 5.6. Based on our physical network topology, each query takes 5.3 hops on average in the network. We repeated the experiment for Basic, Advanced, and ExSPAN for 100 queries each.

Figure 12 shows our experimental results in the form of a CDF of query latency. We observe that both Basic and Advanced have latency numbers that are significantly lower compared to ExSPAN. For example, the mean/median for ExSPAN is 75ms and 74ms respectively, as compared to only 25.5ms and 25ms for Basic. This is approximately a 3X reduction in latency times. The extra overhead is due to ExSPAN's need in processing the larger intermediate tuples. Basic and Advanced avoid this overhead by symbolically rederiving intermediate results during query execution.

6.2 Application #2: DNS Resolution

DNS resolution [12] is an Internet service that translates the requested domain name, such as "www.hello.com", into its corresponding IP address in the Internet. In practice, DNS resolution is performed by DNS nameservers, which are organized into a tree-like structure, where each nameserver is responsible for a domain name (e.g., "hello.com" or ".com"). We used the *recursive name resolution* protocol in DNS, and implemented the protocol as a DELP program (see Appendix A). During the execution of each DELP DNS program, provenance support is enabled so that the history DNS requests can be queried.

We synthetically generated the hierarchical network of DNS name servers. In total, there were 100 name servers, and the maximum tree depth is 27. Our workload consists of clients issuing requests to 38 distinct URLs. In total, DNS requests were issued at a rate of 1000 requests/second. Our topology resembles real-world DNS deployments. Prior work [8] has shown that in reality, the requested domain names satisfy Zipfian distribution. In our experiments, we adopted the same distribution.

6.2.1 Storage of Provenance Trees

Figure 13 shows the provenance storage growth rate for all nameservers in the Domain Name System over a 100 seconds duration. We measure the storage growth of each nameserver by first measuring the growth rate of each 10-second interval, and calculating the average growth rates over all 10 intervals. We observe that ExSPAN has the largest storage growth rate for each node among the three experiments, while Advanced has the lowest storage growth rate. Note that the reduction of storage growth rate in Figure 13 is not as significant as that in the packet forwarding experiments (Figure 8). For example, 80% of nameservers in ExSPAN have storage growth rate less than 476 Kbps. while the rate is 121 Kbps for Advanced. Advanced is four times better than ExSPAN, compared to 11 times in packet forwarding. The reason is that, compared to packet forwarding, we rate the total throughput of incoming events – i.e., packet tuples in packet forwarding and request tuple in DNS resolution – and this causes the storage growth rate at each node using either ExSPAN and Basic to drop as well.

Figure 16 shows the provenance storage growth for all name servers. We record the current storage growth rate at 10-second intervals. In Figure 16, the storage of ExSPAN and Basic grow much faster than that of Advanced. Specifically, the growth rate of ExSPAN, Basic and Advanced are 13.15 Mbps, 11.57 Mbps and 3.81 Mbps respectively, and the storage space at 100 seconds reaches 1.32 GB, 1.16 GB, and 0.38 GB respectively. With the given rates, ExSPAN would fill up a 1TB disk within 21 hours, Basic within 24 hours, and Advanced up to 3 *days*.

Figure 14 shows the storage growth when we increased the number of requested URLs. In this experiment, we fixed the total number of requests at 200, so that when more URLs were added, there would be fewer duplicate requests. In Figure 14, the storage overhead for ExSPAN and Basic remains stable at around 2.5 MB and 2.26 MB respectively. This is because the storage overhead is mostly determined by the number of provenance trees, which is equal to the number of incoming requests (i.e., 200 in this case). For Advanced, the storage grows at a rate of 11.6 Kb per URL. This is expected as we need to maintain one provenance tree for each equivalence class, and the number of equivalence classes grows in proportion to the number of URLs. Similar to our packet forwarding results, despite the storage growth, Advanced still requires significantly less storage compared ExSPAN and Basic. Unless a URL is only requested once (highly unlikely in reality), which represents the worst possible case for Advanced, Advanced always performs better than ExSPAN and Basic.

6.2.2 Network Overhead

Figure 15 shows the bandwidth usage with elapsed time when we continuously sent 100,000 requests to the root nameserver. All three experiments finish within 102 seconds. Throughout the execution, ExSPAN and Basic have similar bandwidth usage, which is stable at around 4.5 MBps. On the other hand, Advanced's bandwidth usage is about 6 MBps, which is about 25% higher than the other two techniques. This is because unlike in the packet forwarding experiments where each packet carries a payload of 500 characters, each DNS request does not have any extra payload. Therefore, the meta-data tagged with each request (e.g., *existFlag*) accounts for a large part of the size of each request, resulting in higher additional bandwidth overhead.

7. RELATED WORK

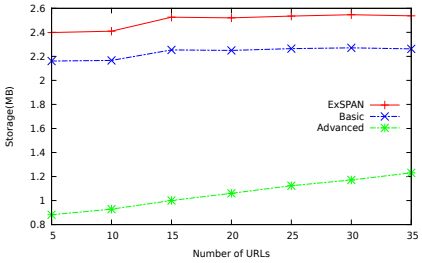

Figure 14: Provenance storage growth with increasing URLs, with 200 requests sent in total.

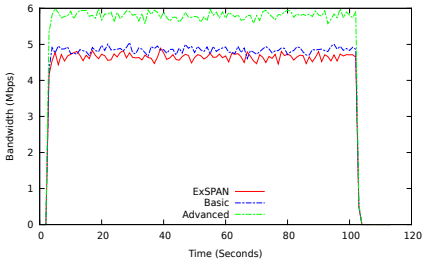

Figure 15: Bandwidth consumption for DNS resolution with 100,000 DNS requests.

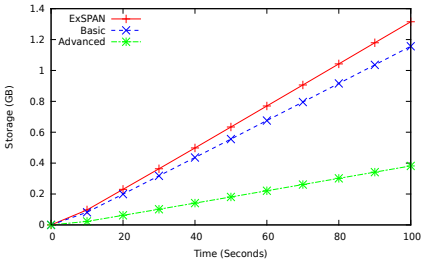

Figure 16: Provenance storage growth with continuous input requests at 1000 requests/second.

Network provenance has been proposed and developed by ExSPAN [27] and DTaP [26]. These two proposals store uncompressed provenance information, laying the foundation for our work. In database literature, a number of works have considered optimization of provenance storage. However, we differ significantly in our design due to the distributed nature of our target environment. We briefly list a few representative bodies of work, and explain our differences.

Woodruff *et al.* [20] reduce storage usage for maintaining fine-grained lineage (i.e., provenance) by computing provenance information dynamically during query time through invertible functions. Their approach tradeoffs storage with accuracy of provenance. On the other hand, our approach requires no such tradeoff, achieving the same level of accuracy as queries on uncompressed provenance trees.

Chapman *et al.* [3] develop a set of factorization algorithms to compress workflow provenance. Their proposal does not consider a distributed setting. For example, node-level factorization (combining identical nodes) requires additional states to be maintained and propagated from node to node during provenance maintenance to resolve potential ambiguities. Maintaining and propagating these states can lead to significant communication overhead in a distributed environment. In contrast, our solution uses the equivalence keys to avoid comparing provenance trees on a node-by-node basis, and hence minimizes communication overhead during provenance maintenance.

Our compression technique implicitly factorizes provenance trees at runtime before removing redundant factors among trees in the same equivalence class. Olteanu *et al.* [15][16] propose factorization of provenance polynomials for conjunctive queries with a new data structure called factorization tree. Polynomial factorization in [16] can be viewed as a more general form of the factorization used in the equivalence-based compression proposed in this paper. If we encode the provenance trees of each packet as polynomials, the general factorization algorithm in [16], with specialized factorization tree, would produce the same factorization result in our setting. Our approach is slightly more efficient, as we can skip the factorization step by directly using the equivalence keys at runtime to group provenance trees for compression. Exploring the more general form of factorization in [16] for provenance of distributed queries is an interesting avenue of future work.

ProQL [9] proposes to save the storage of *single* provenance tree by (1) using primary keys to represent tuples in the provenance, and (2) maintaining one copy for attributes of the same values in a mapping (rule). These techniques could also be applied alongside our online compression algorithm to further reduce storage. ProQL does not consider storage sharing *across* provenance trees. Amsterdamer *et al.* [1] theoretically defines the concept of *core provenance*, which represents derivation shared by multiple equivalent queries. In our scenario, the shared provenance tree of each equivalence class can be viewed as *core provenance*.

Xie *et al.* [23] propose to compress provenance graphs with a hybrid approach combining Web graph compression and dictionary encoding. Zhifeng *et al.* [2] proposes rule-based provenance compression scheme. Their approaches on a high level compresses provenance trees to reduce redundant storage. However, these approaches require knowledge of the *entire* trees prior to compression, which is not practical, if not impossible, for distributed provenance.

Provenance has been applied to network repairing [22, 21, 4] where root-cause analysis is used to identify and fix configuration errors in networks. Network repairing is orthogonal to our work, but can benefit from our compression techniques to reduce the storage of provenance maintenance.

8. CONCLUSION & FUTURE WORK

In this paper, we propose an online, equivalence-based compression scheme for the maintenance of distributed network provenance. Equivalent provenance trees are identified at compile time through static analysis of the declarative program, whereas our runtime maintains only one concrete representative provenance tree for each equivalence class. Our evaluation results show that the compression scheme saves storage significantly, incurs little network overhead, and allows for efficient provenance query. As future work, we plan to extend our compression scheme to provenance trees generated by multiple programs that run concurrently.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback to help improve the paper. We also thank Ling Ding and Yang Li for comments on the work and proof-read of the paper. This work is jointly funded by NSF CNS-1513679, CNS-1218066, CNS-1065130, CNS-1513961, CNS-1453392 and CNS-1513734.

10. REFERENCES

[1] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. *ACM Trans. Database Syst.*, 37(4):30, 2012.

[2] Z. Bao, H. Köhler, L. Wang, X. Zhou, and S. W. Sadiq. Efficient provenance storage for relational queries. In *CIKM*, pages 1352–1361, 2012.

[3] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of ACM SIGMOD*, pages 993–1006, 2008.

[4] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM*, Aug. 2016.

[5] C. Chen, L. Jia, H. Xu, C. Luo, W. Zhou, and B. T. Loo. A program logic for verifying secure routing protocols. In *Proceedings of FORTE*, pages 117–132, 2014.

[6] R. Droms. Dynamic host configuration protocol. 1997. RFC 2131.

[7] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of PODS*, pages 31–40, 2007.

[8] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

[9] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proceedings of ACM SIGMOD*, pages 951–962, 2010.

[10] 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 *Proceedings of ACM SIGMOD*, 2006.

[11] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. In *Communications of the ACM*, 2009.

[12] P. V. Mockapetris. *Domain names - implementation and specification*, Nov. 1987. RFC 1035.

[13] S. C. Muthukumar, X. Li, C. Liu, J. B. Kopena, M. Oprea, and B. T. Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *SIGCOMM (demo)*, 2009.

[14] ns 3 project. Network Simulator 3. http://www.nsnam.org/.

[15] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *Proceedings of TaPP*, 2011.

[16] D. Olteanu and J. Závodný. Factorised representations of query results: size bounds and readability. In *Proceedings of ICDT*, pages 285–298, 2012.

[17] D. C. Plummer. An ethernet address resolution protocol. 1982. RFC 826.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of ACM SIGCOMM*, pages 323–334, 2012.

[19] Robert Ramey. http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/index.html.

[20] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of ICDE*, pages 91–102, 1997.

[21] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of HotNets*, pages 26:1–26:7, 2015.

[22] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proceeding of ACM SIGCOMM*, pages 383–394, 2014.

[23] Y. Xie, K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long. Evaluation of a hybrid approach for efficient provenance storage. *TOS*, 9(4):14, 2013.

[24] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings IEEE INFOCOM*, pages 594–602, 1996.

[25] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of SOSP*, pages 295–310, 2011.

[26] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. G. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. *PVLDB*, 6(2):49–60, 2012.

[27] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of ACM SIGMOD*, pages 615–626, 2010.

APPENDIX

A. DELP FOR DNS RESOLUTION

$$
\begin{array}{ll}
r1 & \mathsf{request}(@RT, URL, HST, RQID) :- \\
 & \quad \mathsf{url}(@HST, URL, RQID). \\
 & \quad \mathsf{rootServer}(@HST, RT). \\
r2 & \mathsf{request}(@SV, URL, HST, RQID) :- \\
 & \quad \mathsf{request}(@X, URL, HST, RQID), \\
 & \quad \mathsf{nameServer}(@X, DM, SV). \\
 & \quad \mathsf{f\_isSubDomain}(DM, URL) == true. \\
r3 & \mathsf{dnsResult}(@X, URL, IPADDR, HST, RQID) :- \\
 & \quad \mathsf{request}(@X, URL, HST, RQID), \\
 & \quad \mathsf{addressRecord}(@X, URL, IPADDR). \\
r4 & \mathsf{reply}(@HST, URL, IPADDR, RQID) :- \\
 & \quad \mathsf{dnsResult}(@X, URL, IPADDR, HST, RQID).
\end{array}
$$

Figure 17: DELP for DNS resolution

Figure 17 shows the DELP encoding of the recursive DNS resolution. The program is composed of four rules. Rule *r1* forwards a DNS request of ID *RQID* to the root nameserver *RT* for resolution. The request is generated by the host *HST* for the URL *URL*. Rule *r2* is triggered when a nameserver X receives a DNS request for *URL*, but has delegated the resolution of sub-domain *DM* corresponding to *URL* to another nameserver *SV*. Rule *r2* then forwards the DNS request to *SV* for further DNS resolution. Rule *r3* generates a DNS resolution result containing the IP address *IPADDR* corresponding to the requested *URL*, when *URL* matches an address record on the nameserver X. Finally, Rule *r4* is responsible for returning the DNS result to the requesting host *HST*.

B. DELP PROGRAMS

We present the syntax of DELP's programs in Figure B. A DELP program *DQ* is composed of an ordered list of rules. Each rule r consists of a head *hd* and a body *body*. A rule head is a relation, while a rule body consists of a list of body elements which are either relations, assignments or constraints. Intuitively, a DELP rule specifies that the head tuple is derivable if all the body tuples are derivable and all the constraints are satisfied.

$$
\begin{array}{llll}
\textit{DELP Program} & DQ & ::= & [r_1, \cdots, r_n] \\
\textit{DELP Rule} & r & ::= & hd \text{ :- } body \\
\textit{Rule Head} & hd & ::= & ev \,|\, res \,|\, P \\
\textit{Rule Body} & body & ::= & ev, B_1, \cdots, B_n \mathsf{a}_1, \cdots, \mathsf{a}_m, c_1, \cdots, c_N
\end{array}
$$

Figure 18: Syntax of DELP programs

We explain the relations that appear in each DELP rule in Figure B.

First we define some constructs that are used to specify the relations. Terms are either variables represented by x, or constants represented by c. Each DELP rule in *DQ* has a unique identifier *rID* for reference.

Each tuple in the program has a location specifier to declare its location. The location specifier is the first attribute in a relation and is represented by $@\eta$. We prepend the first attribute of a relation with the "@" symbol as a reminder that it represents the location of the relation. In particular, we write ι to refer to a concrete location specifier and ℓ to denote a variable representing a location specifier.

All relations in the body of a rule must reside on the same node. However, the rule head can be location on a different node from the rule body. In this case, when the rule is executed, the derived head tuple is sent across the network to the remote node. We discuss the operational semantics of DELP in further detail in appendix D.

We define a declaration Γ to describe types of relations that can appear in *DQ*. Furthermore, Γ also stores the primary keys for each tuple, which always includes the location specifier.

DELP distinguishes between slow-changing tuples and fast-changing tuples. Slow-changing tuples are assumed to be populated upon system initialization and do not change during a fixpoint computation.

Slow-changing relations have type "slow" to specify that they do not change during a fixpoint execution. We write B to refer to a slow-changing tuple and $b(@\ell_b, \vec{x}_b)$ to specify that a slow-changing relation has arguments $@\ell_b, \vec{x}_b$.

A relation of type "fast" refers to a fast-changing relation of program that does not appear in the body of rule r_1. We write P to refer to a fast-changing tuple and $p(@\ell_p, \vec{x}_p)$ to specify that a fast-changing relation has arguments $@\ell_p, \vec{x}_p$. In some cases, we may also use Q and $q(@\ell_q, \vec{x}_q)$ to denote a fast-changing tuple.

A relation of type "event" refers to the the fast-changing event relation in the first rule of the program. When an event tuple arrives on a node, it triggers program execution. We write ev to refer to an event tuple and $e(@\ell_e, \vec{x}_e)$ to specify that the event relation has arguments $@\ell_e, \vec{x}_e$.

Finally, a relation of type "interest" refers to a fast-changing relation that the user additionally specifies as a relation of interest. Our compression algorithm stores the provenance of rules fired, but normally omits storing tuple provenance to save space. The user must specify a fast-changing relation is specified to be a relation of interest, in order for our algorithm to store corresponding tuple provenance. We write res to refer to a tuple of a relation of interest and $r(@\ell_r, \vec{x}_r)$ to specify that the relation of interest has arguments $@\ell_r, \vec{x}_r$.

Each rule in DQ consist of one fast-changing relation (that may be of type event, fast, or interest) that triggers execution of the rule when it arrives on a node ι and joins on the other slow-changing relations (of type slow) present in local database of node ι.

Terms	t	::=	$x \mid c$
Rule Identifier	rID	::=	rID
Location Specifier	η	::=	$\iota \mid \ell$
Declaration	Γ	::=	$e \mapsto [equivalence\_keys, K], [tuple, \mathsf{event}], [primary\_keys, j_1 :: \cdots :: j_n]$
			$\mid p \mapsto [tuple, \mathsf{fast}], [primary\_keys, j_1 :: \cdots :: j_n]$
			$\mid r \mapsto [tuple, \mathsf{interest}], [primary\_keys, j_1 :: \cdots :: j_n]$
			$\mid b \mapsto [tuple, \mathsf{slow}], [primary\_keys, j_1 :: \cdots :: j_n]$
Event relation	ev	::=	$e(@\eta, \vec{t})$
Slow-changing relation	B	::=	$b(@\eta, \vec{t})$
Derived relation	P	::=	$p(@\eta, \vec{t})$
Relation of interest	res	::=	$r(@\eta, \vec{t})$

Figure 19: Syntax of relations that appear in DELP rules

Besides relations, rules may also contain assignments or constraints. Assignments are used to specify a fresh variable in the head tuple. The are computed either using a deterministic function that takes variables in the body relations as inputs and outputs the value of the fresh variable, or returned by an arithmetic expression composed from variables in the body relations. Finally, constraints are used to restrict the tuples that are used to execute a rule.

Assignment	a	::=	$t := \textsc{Fun}(\vec{t}) \mid t := ar$
Arithmetic operator	aop	::=	$+ \mid - \mid \times \mid \div$
Arithmetic expression	ar	::=	$t \mid ar_1\ aop\ ar_2$
Arithmetic Operator	aop	::=	$+ \mid - \mid \times \mid \div$
Comparator	cop	::=	$\geq \mid > \mid = \mid < \mid \leq$
Binary Operator	bop	::=	$\wedge \mid \vee \mid \supset$
Constraint	c	::=	$ar_1\ cop\ ar_2 \mid c_1\ bop\ c_2 \mid \neg c$

Figure 20: Syntax of DELP rules

C. CORRECTNESS OF STATIC ANALYSIS

Given a DELP program DQ, two provenance trees tr and tr' generated by bottom-up execution of DQ are said to be equivalent if they are structurally identical – i.e., they trigger an identical sequence of rules – and join with identical slow-changing tuples in each rule. Thus, tr and tr' only differ at two nodes: (1) the root node that represents the output tuple and (2) the input event tuple. We denote the minimal set of attributes K in the input event relation whose values determine the provenance trees as *equivalence keys*. In Section 5.2, we defined a static analysis algorithm to identify the equivalence keys of the input event relation. In this section, we show that our static analysis algorithm is correct.

C.1 Definitions

We define additional constructs we used to prove that our static analysis is correct. We write $\mathcal{DB}$ to denote a set of slow-changing tuples and derived fast-changing tuples corresponding to relations in DQ. We write $DQ, ev, \mathcal{DB} \vDash tr : P$ to mean that tr is a derivation tree for tuple P using program DQ and materialized tuples $\mathcal{DB}$. Tuple P is the root of tr and event tuple ev is the left-most leaf of tr. A provenance tree tr:P represents the full derivation of the derived tuple P. The semantics of DELP programs are bottom up, so in the base case only one rule was fired to derive P. This rule was the first rule of the DELP program has unique identifier rID and was triggered by event tuple ev to join on slow-changing tuples $B_1, \cdots, B_n$. In the inductive case, tuple Q is a fast-changing tuple that is not an event tuple. It triggered execution of a rule with unique identifier rID and joined with slow-changing tuples $B_1, \cdots, B_n$ to derive tuple P.

$$\begin{array}{lll} \textit{Materialized tuples } \mathcal{DB} & ::= & \cdot \mid \mathcal{DB}, P \\ \textit{Provenance Tree} \quad tr & ::= & \langle rID, P, ev, B_1 :: \cdots :: B_n \rangle \mid \langle rID, P, tr{:}Q, B_1 :: \cdots :: B_n \rangle \end{array}$$

C.2 Properties

Next, we define several rules that we will use to prove the correctness of equivalence keys.

Given a rule $rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots \in DQ$, we define rules the capture the ways in which attributes of trigger tuple $q(\vec{x}_q)$ are connected to slow-changing tuples We write $DQ \vdash \mathsf{joinSAttr}(q{:}i)$ to mean that the i^{th} attribute of q is The rules are:

Rule JOIN-SLOW**.** If an attribute on a fast-changing relation in the body of a rule is the same as an attribute on a slow-changing relation in the body, then that fast-changing attribute joins with a slow-changing attribute.

Rule JOIN-FUNC-ATTR**.** If an attribute on a fast-changing relation in the body of a rule is the same as an attribute that appears on the right-hand side of an assignment, then that fast-changing attribute joins with a slow-changing attribute.

Rule JOIN-ARITH-LEFT**.** If an attribute on a fast-changing relation in the body of a rule is the same as an attribute that appears on the left-hand side of an arithmetic constraint, then that fast-changing attribute joins with a slow-changing attribute.

Rule JOIN-ARITH-RIGHT**.** If an attribute on a fast-changing relation in the body of a rule is the same as an attribute that appears on the right-hand side of an arithmetic constraint, then that fast-changing attribute joins with a slow-changing attribute.

$\boxed{DQ \vdash \mathsf{joinSAttr}(p{:}i)}$

$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), b_1(\vec{x}_{b1}), \cdots, b_k(\vec{x}_{bk}), \cdots, b_n(\vec{x}_{bn}), \cdots \in DQ \qquad q{:}i = b_k{:}j}{DQ \vdash \mathsf{joinSAttr}(q{:}i)}\ \textsc{Join-Slow}$$

$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, F_i : y := F(\vec{z}), \cdots \in DQ \qquad q{:}j = \vec{z}{:}k}{DQ \vdash \mathsf{joinSAttr}(q{:}j)}\ \textsc{Join-Func-Attr}$$

$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, a_L(\vec{x}_{aL}) bop\ a_R(\vec{x}_{aR}), \cdots \in DQ \qquad a_L.j = q.i}{DQ \vdash \mathsf{joinSAttr}(q.i)}\ \textsc{Join-Arith-Left}$$

$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, a_L(\vec{x}_{aL}) bop\ a_R(\vec{x}_{aR}), \cdots \in DQ \qquad a_R.j = q.i}{DQ \vdash \mathsf{joinSAttr}(q.i)}\ \textsc{Join-Arith-Right}$$

Given a rule $rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots \in DQ$, rule JOIN-HEAD states that head tuple $p(\vec{x}_p)$ is connected to the fast changing tuple $q(\vec{x}_q)$ in the body if they share identical values for their attributes.

$\boxed{DQ \vdash \mathsf{joinFAttr}(q{:}i, p{:}j)}$

$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots \in DQ \qquad p{:}j = q{:}i}{DQ \vdash \mathsf{joinFAttr}(q{:}i, p{:}j)}\ \textsc{Join-Head}$$

We write $DQ \vdash \mathsf{connected}(e{:}i, p{:}j)$ to mean that the i^{th} attribute of the input event relation e is connected to the j^{th} attribute of the fast-changing relation p.

Rule Connected-Slow**.** If the i^{th} attribute of the input event relation e joins with the j^{th} attribute of the fast-changing relation p, then $p{:}j$ is connected to $e{:}i$

Rule Connected-Join**.** If the i^{th} attribute of the input event relation e is connected to the j^{th} attribute of the fast-changing relation q, and if the j^{th} attribute of the fast-changing relation q joins with the k^{th} attribute the fast-changing relation p, then the i^{th} attribute of the input event relation e is connected to the k^{th} attribute the fast-changing relation p.

$\boxed{DQ \vdash \mathsf{connected}(q{:}i, p{:}j)}$

$$\frac{DQ \vdash \mathsf{joinFAttr}(e{:}i, p{:}j)}{DQ \vdash \mathsf{connected}(e{:}i, p{:}j)}\ \textsc{Connected-Slow} \qquad \frac{DQ \vdash \mathsf{connected}(e{:}i, q{:}j) \qquad DQ \vdash \mathsf{joinFAttr}(q{:}j, p{:}k)}{DQ \vdash \mathsf{connected}(e{:}i, p{:}k)}\ \textsc{Connected-Join}$$

An attribute $e{:}i$ of event tuple e is in the set of equivalence keys ($DQ \vdash e{:}i \in \mathit{equi\_attr}$) if it is connected to a slow changing tuple.

Rule Equi-Direct**.** $e{:}i$ is in the set of equivalence keys when it shares attributes with a slowing changing tuple within a rule

Rule Equi-Reachable**.** Alternatively, if $e{:}i$ is connected to an attribute of a fast-changing tuple $q{:}j$, and $q{:}j$ joins with an attribute of some slow changing tuple, then $e{:}i$ is also in the set of equivalence keys.

$\boxed{DQ \vdash e{:}i \in \mathit{equi\_attr}}$

$$\frac{DQ \vdash \mathsf{joinSAttr}(e{:}i)}{DQ \vdash e{:}i \in \mathit{equi\_attr}}\ \textsc{Equi-Direct} \qquad \frac{DQ \vdash \mathsf{joinSAttr}(q{:}j) \qquad DQ \vdash \mathsf{connected}(e{:}i, q{:}j)}{DQ \vdash e{:}i \in \mathit{equi\_attr}}\ \textsc{Equi-Reachable}$$

Two provenance trees tr_1 and tr_2 that store the provenance of two separate executions of DQ are equivalent ($tr_1 \sim_K tr_2$) if their input event tuples are equivalent and they differ only at derived tuples.

Rule $\sim_K$-Base**.** In the base case, only one rule has been fired. If the input event tuples that triggered both executions is equivalence, and both executions used the same slow-changing tuples to fire that rule, then their derivation trees tr_1 and tr_2 are the same.

Rule $\sim_K$-Inductive**.** If tr and tr' are equivalent derivation trees for tuples Q and Q' respectively, the resultant derivation trees after Q and Q' have been used to fire one subsequent rule using the same slow-changing tuples are again equivalent.

$\boxed{tr_1 \sim_K tr_2}$

$$\frac{ev \sim_K ev'}{\langle rID, P, ev, B_1 :: \cdots :: B_n\rangle \sim_K \langle rID, P', ev', B_1 :: \cdots :: B_n\rangle}\ \sim_K\textsc{-Base}$$

$$\frac{tr \sim_K tr'}{\langle rID, P, tr, B_1 :: \cdots :: B_n\rangle \sim_K \langle rID, P', tr', B_1 :: \cdots :: B_n\rangle}\ \sim_K\textsc{-Inductive}$$

C.3 Lemmas and Proofs

Correctness of equivalence keys (Theorem 1) shows that given a DELP program DQ, the equivalence keys that our method returns is able to determine the equivalence class of any incoming event tuple. We always include the attribute indicating the input location of ev in the equivalence keys to prevent the input event tuples on different network nodes from having identical equivalence keys. Instead of directly proving Correctness of equivalence keys (Theorem 1), we prove a stronger lemma about provenance trees that gives us Theorem 1 as a corollary.

This theorem states that given two equivalent input event tuples ev_1 and ev_2 w.r.t. K, where K is identified by our static analysis algorithm, and that ev_1 generates provenance tree tr_1, we can construct a tr_2 for ev_2 such that tr_1 and tr_2 are equivalent – i.e., they share the same structure and slow changing tuples; further, the result (query) tuples of these two trees only differ in attributes that connect to attributes of the input event tuple that are not part of the equivalent key. This additional condition allows for an inductive proof over the structure of the tree.

C.3.1 Correctness of equivalence keys

Theorem 1 (Correctness of equivalence keys).
$\textsc{GetEquiKeys}(DQ, \Gamma) = K$
and $e(@\iota, t_{e_1}, \cdots, t_{e_m}) \sim_K e(@\iota, s_{e1}, \cdots, s_{em})$
and $DQ, \mathcal{DB}, e(@\iota, t_{e_1}, \cdots, t_{e_m}) \vDash tr_1 : p(t_1, \cdots, t_n)$
implies
$\exists tr_2 : p(s_1, \cdots, s_n)$ s.t.
$DQ, \mathcal{DB}, e(@\iota, s_{e1}, \cdots, s_{em}) \vDash tr_2 : p(s_1, \cdots, s_n)$
and $tr_1 : p(t_1, \cdots, t_n) \sim_K tr_2 : p(s_1, \cdots, s_n)$.

Proof.
Assume
(1) $\textsc{GetEquiKeys}(DQ, \Gamma) = K$
(2) $e(@\iota, t_{e_1}, \cdots, t_{e_m}) \sim_K e(@\iota, s_{e1}, \cdots, s_{em})$
(3) $DQ, \mathcal{DB}, e(@\iota, t_{e_1}, \cdots, t_{e_m}) \vDash tr_1 : p(t_1, \cdots, t_n)$

By (1), (2), and (3) we apply Correctness of equivalence keys (Strong) to obtain that
(4) $\exists tr_2{:}p(s_1, \cdots, s_n)$ s.t.
$DQ, \mathcal{DB}, e(@\iota, s_{e1}, \cdots, s_{em}) \vDash tr_2 : p(s_1, \cdots, s_n)$
and $tr_1 : p(t_1, \cdots, t_n) \sim_K tr_2 : p(s_1, \cdots, s_n)$
and $\forall i \in [1, n]$, $t_i \neq s_i$ implies $\exists \ell$ s.t. $DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i)$ and $\ell \notin K$

By (4),
The conclusion follows

□

C.3.2 Correctness of equivalence keys (Strong)

Lemma 2 (Correctness of equivalence keys (Strong)).
$\textsc{GetEquiKeys}(DQ, \Gamma) = K$
and $ev_1 \sim_K ev_2$
and $DQ, \mathcal{DB}, ev_1 \vDash tr_1 : p(t_1, \cdots, t_n)$
implies
$\exists tr_2 : p(s_1, \cdots, s_n)$ s.t.
$DQ, \mathcal{DB}, ev_2 \vDash tr_2 : p(s_1, \cdots, s_n)$
and $tr_1 : p(t_1, \cdots, t_n) \sim_K tr_2 : p(s_1, \cdots, s_n)$
and $\forall i \in [1, n]$,
$t_i \neq s_i$
implies
$\exists \ell$ s.t. $DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i)$ and $\ell \notin K$.

Proof.
By induction over the structure of tr'_1.

Base Case: $tr'_1{:}p(t_{p1}, \cdots, t_{pn}) = \langle e(t_{e1}, \cdots, t_{em}), \langle rID, p(\vec{t}_{p1}), b_1(\vec{t}_{b1}) :: \cdots :: b_N(\vec{t}_{bN}) :: \mathsf{nil} \rangle {:} p(t_{p1}, \cdots, t_{pn})$

We show that $\exists tr_2{:}p(s_{p1}, \cdots, s_{pn})$ **s.t.** $\boldsymbol{DQ}, \mathcal{DB}, e(s_{e1}, \cdots, s_{em}) \vdash tr_2{:}p(s_{p1}, \cdots, s_{pn})$**.**
By the assumptions,
(i1) $\exists r \in DQ$ s.t.

$$r = \quad rID \quad p(x_{p1}, \cdots, x_{pn}) \quad :- \quad \begin{array}{l} e(x_{e1}, \cdots, x_{em}), \\ b_1(\vec{x}_{b1}), \cdots, b_N(\vec{x}_{bN}), \\ F_1 : y_1 := \textsc{Fun1}(\vec{z}_1), \cdots, F_M : y_M := \textsc{FunM}(\vec{z}_M), \\ a_{L1}(\vec{x}_{aL1})\ bop\ a_R(\vec{x}_{aR1}), \cdots, a_{L\Lambda}(\vec{x}_{aL\Lambda})\ bop\ a_{R\Lambda}(\vec{x}_{aR\Lambda}) \end{array} \in DQ$$

Define:
$\sigma_1 \triangleq \{t_{p1}/x_{p1}, \cdots, t_{pn}/x_{pn}\} \cup \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
$\sigma'_1 \triangleq \sigma_1 \cup \bigcup_{i=1}^{M} [\sigma_1(y_i)/\vec{y_i}]$
Since $r\sigma'_1{:}p(x_{p1}, \cdots, x_{pn})\sigma'_1 = tr_1{:}p(t_{p1}, \cdots, t_{pn})$,
σ'_1 is a well-formed substitution

Define:
$\sigma_2 \triangleq \{s_{p1}/x_{p1}, \cdots, s_{pn}/x_{pn}\} \cup \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
$\sigma'_2 \triangleq \sigma_2 \cup \bigcup_{i=1}^{M} [\sigma_2(y_i)/\vec{y_i}]$
By definition,
$\sigma'_1 \left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right) = \sigma'_2 \left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right)$

We show that σ_2' is a well-formed substitution.
Pick any $[t_a/x_a], [t_b/x_b] \in \sigma_2'$ s.t. $x_a = x_b$.
Our goal is to show that $t_a = t_b$

Cases to consider:
(A) $[t_a/x_a], [t_b/x_b] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$
(B) $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
(C) $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/\vec{y_i}]$
(D) $[t_a/x_a] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$ and $[t_b/x_b] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
(E) $[t_a/x_a] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$ and $[t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/\vec{y_i}]$
(F) $[t_a/x_a] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$ and $[t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/y_i]$

Case A: $[t_a/x_a], [t_b/x_b] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$
By assumption,
$\exists i \in \{x_{e1}, \cdots, x_{em}\}$ s.t. $x_a = e{:}i = x_b$
By the above,
$\sigma_2'(x_a) = t_a = \sigma_2'(e{:}i) = t_b = \sigma_2'(x_b)$

Case B: $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
Similar argument to Case A

Case C: $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/\vec{y_i}]$
Similar argument to Case A

Case D: $[t_a/x_a] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$ **and** $[t_b/x_b] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$
Subcase I: $x_a = e{:}i$ *and* $e{:}i \in K$
Since $x_a = x_b$,
$e{:}i = x_b$
By the above, we apply Join-Base and obtain:
$DQ \vdash \mathsf{joinSAttr}(e{:}i)$
Since $DQ \vdash \mathsf{joinSAttr}(e{:}i)$, we apply Equi-Direct to obtain:
$DQ \vdash e{:}i \in K$
Since $DQ \vdash \mathsf{joinSAttr}(e{:}i)$
and $e(t_{e1}, \cdots, e_{tm}) \sim_K e(s_{e1}, \cdots, s_{tm})$
and $x_b \in \bigcup_{i=1}^{N} \vec{x}_{bi}$,
$\sigma_1'(x_a) = \sigma_2'(x_a)$ and $\sigma_2'(x_b) = \sigma_1'(x_b)$
Since $\sigma_1'(x_a) = \sigma_1'(x_b)$ as σ_1' is well-formed,
$\sigma_2'(x_a) = t_a = \sigma_2'(x_b) = t_b$

Subcase II: $x_a = e{:}i$ *and* $e{:}i \notin K$
Since $x_a = x_b$,
$e{:}i = x_b$
By the above, we apply Join-Base and obtain:
$DQ \vdash \mathsf{joinSAttr}(e{:}i)$
Since $DQ \vdash \mathsf{joinSAttr}(e{:}i)$, we apply Equi-Direct to obtain:
$DQ \vdash e{:}i \in K$
This contradicts our assumption that $e{:}i \notin K$

Case E: $[t_a/x_a] \in \{s_{e1}/x_{e1}, \cdots, s_{em}/x_{em}\}$ **and** $[t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/y_i]$
Since $x_b \in \bigcup_{i=1}^{M} y_i$
and $\left(\bigcup_{i=1}^{M} y_i\right) \cap \left(\{x_{e1}, \cdots, x_{em}\} \cup \bigcup_{i=1}^{N} \vec{x}_{bi}\right) = \emptyset$
and $x_a \in \left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right)$
$x_a \neq x_b$
This contradicts our assumption that $x_a = x_b$

Case F: $[t_a/x_a] \in \bigcup_{i=1}^{N} [\vec{t}_{bi}/\vec{x}_{bi}]$ **and** $[t_b/x_b] \in \bigcup_{i=1}^{M} [\sigma_2(y_i)/y_i]$
Similar argument to Case E

We show that $tr_1{:}p(t_{p1}, \cdots, t_{pn}) \sim_K tr_2{:}p(s_{p1}, \cdots, s_{pn})$
Define
$tr_{abs} = \langle rID, p(x_{p1}, \cdots, x_{pn}), e(x_{e1}, \cdots, x_{em}), b_1(\vec{x}_{b1}) :: \cdots :: b_N(\vec{x}_{bN}) \rangle$
Since σ_2' is well-formed, we define:
$tr_2 \triangleq \sigma_2'(tr_{abs})$
and $\sigma_1'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right) = \sigma_2'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right)$
and $e(t_{e1}, \cdots, t_{em}) \sim_K e(s_{e1}, \cdots, s_{em})$

and $tr_1 = \sigma_1'(tr_{abs})$
we apply $\sim_K$-BASE to obtain:
$tr_1 \sim tr_2$

We show that $\forall i \in [1,n]$**,** $t_{pi} \neq s_{pi}$ **implies** $\exists \ell$ **s.t.** $\boldsymbol{DQ} \vdash \mathsf{connected}(e{:}\ell, p{:}i)$ **and** $\ell \notin K$
Pick any $i \in [1,n]$.
Assume $\sigma_1'(p{:}i) \neq \sigma_2'(p{:}i)$.

By definition of σ_1' and σ_2',
$\forall i \in [1,|e|]$, $e{:}i \in K$ implies $\sigma_1'(e{:}i) = \sigma_2'(e{:}i)$
and $\sigma_1'\left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right) = \sigma_2'\left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right)$
and $\forall j \in [1,M]$, $\vec{z}_j \subseteq \left(\bigcup_{i=1}^{N} \vec{x}_{bi}\right)$ implies $\sigma_1'(y_j) = \sigma_2'(y_j)$

Since $\sigma_1'(p{:}i) \neq \sigma_2'(p{:}i)$
and $\vec{x}_p \subseteq \left(\vec{x}_e \cup \bigcup_{i=1}^{N} \vec{x}_{bi} \cup \bigcup_{i=1}^{M} y_i\right)$,
at least one of these cases hold:
Case A: $\exists j \in [1,|e|]$ s.t. $e{:}j = p{:}i$ and $e{:}j \notin K$
Case B: $\exists j \in [1,M]$ s.t. $p{:}i = \vec{y}_j = \textsc{FunJ}(\vec{z}_j)$ and $\sigma_1'(\vec{z}_j \cap \vec{x}_e) \neq \sigma_2'(\vec{z}_j \cap \vec{x}_e)$

Case A: $\exists j \in [1,|e|]$ *s.t.* $e{:}j = p{:}i$ *and* $e{:}j \notin K$
By assumption $\exists j \in [1,|e|]$ s.t. $e{:}j = p{:}i$,
$DQ \vdash \mathsf{connected}(e{:}j, p{:}i)$
By assumption,
$e{:}j \notin K$

Case B: $\exists j \in [1,M]$ *s.t.* $p{:}i = y_j = \textsc{FunJ}(\vec{z}_j)$ *and* $\sigma_1'(\vec{z}_j \cap \vec{x}_e) \neq \sigma_2'(\vec{z}_j \cap \vec{x}_e)$
By the assumptions,
$\forall \ell \in [1,|e|]$ s.t. $e{:}\ell \in \vec{z}_j$ and $\sigma_1'(e{:}\ell) \neq \sigma_2'(e{:}\ell)$
By the above, $\exists k \in [1,|\vec{z}_j|]$ s.t. $e{:}\ell = \vec{z}_j{:}k$, thus by JOIN-FUNC-ATTR,
$DQ \vdash \mathsf{joinSAttr}(e{:}\ell)$
Since $DQ \vdash \mathsf{joinSAttr}(e{:}\ell)$, by EQUI-DIRECT,
$e{:}\ell \in K$
Since $\forall \ell \in [1,|e|]$ s.t. $e{:}\ell \in \vec{z}_j$, $e{:}\ell \in K$
and $\forall \vec{z}_j{:}k$ s.t. $\vec{z}_j \notin \vec{x}_e$, $\vec{z}_j{:}k \in \bigcup_{i=1}^{n} \vec{x}_{bi}$,
$\sigma_1'(\vec{z}_j) = \sigma_2'(\vec{z}_j)$
By the above,
$\sigma_1'(y_j) = \sigma_2'(y_j)$
Since $p{:}i = y_j$,
this contradicts the assumption that $\sigma_1'(p{:}i) \neq \sigma_2'(p{:}i)$

Inductive Case: $tr_1{:}p(t_{p1}, \cdots, t_{pN}) = \langle rID, p(t_{p1}, \cdots, t_{pN}), tr_{q,1}{:}q(t_{q1}, \cdots, t_{qM}), b_1(\vec{t}_{b1}) :: \cdots :: b_n(\vec{t}_{bn})\rangle$
By assumptions,
(i1) $\exists r \in DQ$ s.t.

$$r = \quad rID \quad p(x_{p1}, \cdots, x_{pN}) \quad :\text{-} \quad \begin{array}{l} q(x_{q1}, \cdots, x_{qM}), \\ b_1(\vec{x}_{b1}), \cdots, b_n(\vec{x}_{bn}), \\ F_1 : y_1 := \textsc{Fun1}(\vec{z}_1), \cdots, F_m : y_m := \textsc{FunM}(\vec{z}_m), \\ a_{L1}(\vec{x}_{aL1}) \ bop \ a_R(\vec{x}_{aLR}), \cdots, a_{L\ell}(\vec{x}_{aL\ell}) \ bop \ a_{R\ell}(\vec{x}_{aR\ell}) \end{array} \in DQ$$

Define:
$\sigma_1 \triangleq \{t_{q1}/x_{q1}, \cdots, t_{qM}/x_{qM}\} \cup \bigcup_{i=1}^{n} [\vec{t}_{bi}/\vec{x}_{bi}]$
$\sigma_1' \triangleq \sigma_1 \cup \bigcup_{i=1}^{m} [\sigma_1(y_i)/\vec{y_i}]$
Since $\sigma_1'(r) = tr_1$,
σ_1' is a well-formed substitution
Pick any $tr_{q,2}{:}q(s_{q1}, \cdots, s_{qM})$ s.t. $tr_{q,1}{:}q(t_{q1}, \cdots, t_{qM}) \sim_K tr_{q,2}{:}q(s_{q1}, \cdots, s_{qM})$.
By the induction hypothesis,
$\forall \ell \in [1,M]$, $t_{qi} \neq s_{qi}$ implies $\exists \ell \in [1,|e|]$ s.t. $DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i)$ and $\ell \notin K$

Define:
$\sigma_2 \triangleq \{s_{q1}/x_{q1}, \cdots, s_{qM}/x_{qM}\} \cup \bigcup_{i=1}^{n} [\vec{t}_{bi}/\vec{x}_{bi}]$
$\sigma_2' \triangleq \sigma \cup \bigcup_{i=1}^{m} [\sigma_2(y_i)/\vec{y_i}]$
By definition,
$\sigma_1'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right) = \sigma_2'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right)$

We show that σ_2' is a well-formed substitution.
Pick any $[t_a/x_a], [t_b/x_b] \in \sigma_2'$ s.t. $x_a = x_b$.
Our goal is to show that $t_a = t_b$

Cases to consider:
(A) $[t_a/x_a], [t_b/x_b] \in \{s_{q1}/x_{q1}, \cdots, s_{qM}/x_{qM}\}$
(B) $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{n} [\vec{t}_{bi}/\vec{x}_{bi}]$
(C) $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{m} [\sigma_2(y_i)/\vec{y_i}]$
(D) $\exists i \in [1, M]$ s.t. $[t_a/x_a] = [s_{qi}/q{:}i]$ and $\exists j \in [1, n]$, $\exists k \in [1, |b_j|]$ s.t. $[t_b/x_b] = [\sigma_2(b_j{:}k)/b_j{:}k]$
(E) $\exists i \in [1, M]$ s.t. $[t_a/x_a] = [s_{qi}/x_{qi}]$ and $\exists j \in [1, m]$ s.t. $[t_b/x_b] = [\sigma_2(y_j)/y_j]$
(F) $\exists j \in [1, n]$, $\exists k \in [1, |b_j|]$ s.t. $[t_b/x_b] = [\sigma_2(b_j{:}k)/b_j{:}k]$ and $\exists j \in [1, m]$ s.t. $[t_b/x_b] = [\sigma_2(y_j)/y_j]$

Case A: $[t_a/x_a], [t_b/x_b] \in \{s_{q1}/x_{q1}, \cdots, s_{qM}/x_{qM}\}$
By assumption,
$\exists i \in [1, |q|]$ s.t. $[t_a/x_a] = [s_{qi}/x_{qi}]$
Since $x_a = x_{qi}$ and $x_a = x_b$ and $[t_b/x_b] \in \{s_{q1}/x_{q1}, \cdots, s_{qM}/x_{qM}\}$,
$[t_b/x_b] = [s_{qi}/x_{qi}]$
Therefore $t_a = s_{qi} = t_b$

Case B: $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{n} [\vec{t}_{bi}/\vec{x}_{bi}]$
Similar argument to Case A

Case C: $[t_a/x_a], [t_b/x_b] \in \bigcup_{i=1}^{m} [\sigma_2(y_i)/\vec{y_i}]$
Similar argument to Case A

Case D: $\exists i \in [1, M]$ **s.t.** $[t_a/x_a] = [s_{qi}/q{:}i]$ **and** $\exists j \in [1, n]$**,** $\exists k \in [1, |b_j|]$ **s.t.** $[t_b/x_b] = [\sigma_2(b_j{:}k)/b_j{:}k]$
Assume for contradiction that $t_a \neq t_b$.
Since σ_1' is well-formed,
$\sigma_1'(q{:}i) = t_{qi} = \sigma_1'(b_j{:}k)$
By definition of σ_2',
$\sigma_2'(q{:}i) = s_{qi} = t_a$
$\sigma_2'(b_j{:}k) = t_b$
Since $\sigma_1'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right) = \sigma_2'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right)$,
$t_{qi} \neq s_{qi}$
By the induction hypothesis,
$\exists \ell \in [1, |e|]$ s.t. $DQ \vdash \mathsf{connected}(e{:}\ell, q{:}i)$ and $q{:}i \notin K$
Since $q{:}i = x_a = x_b = b_j{:}k$, by Join-Base,
$DQ \vdash \mathsf{joinSAttr}(q{:}i)$
Given $DQ \vdash \mathsf{connected}(e{:}\ell, q{:}i)$ and $DQ \vdash \mathsf{joinSAttr}(q{:}i)$, by Equi-Reachable,
$e{:}\ell \in K$
This contradicts the earlier statement that $q{:}i \notin K$

Case E: $\exists i \in [1, M]$ **s.t.** $[t_a/x_a] = [s_{qi}/x_{qi}]$ **and** $\exists j \in [1, m]$ **s.t.** $[t_b/x_b] = [\sigma_2(y_j)/y_j]$
By assumption, $y_j \notin x_{q1} :: \cdots :: x_{qM}$, thus
$x_a = x_{qi} \neq y_j = x_b$
Therefore $x_a \neq x_b$ contradicting our earlier assumption

Case F: $\exists j \in [1, n]$**,** $\exists k \in [1, |b_j|]$ **s.t.** $[t_b/x_b] = [\sigma_2(b_j{:}k)/b_j{:}k]$ **and** $\exists j \in [1, m]$ **s.t.** $[t_b/x_b] = [\sigma_2(y_j)/y_j]$
By assumption, $y_j \notin x_{q1} :: \cdots :: x_{qM}$, thus
$x_a = x_{qi} \neq y_j = x_b$
Therefore $x_a \neq x_b$ contradicting our earlier assumption

Since σ_2' is well-formed, we define:
$tr_2 \triangleq \sigma_2'(\langle rID, p(x_{p1}, \cdots, x_{pN}), tr_{q,2}{:}q(x_{q1}, \cdots, x_{qM}), b_1(\vec{x}_{b1}) :: \cdots :: b_n(\vec{x}_{bn})\rangle)$
Given that $\sigma_1'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right) = \sigma_2'\left(\bigcup_{i=1}^{n} \vec{x}_{bi}\right)$
and $tr_{q,1}{:}q(t_{q1}, \cdots, t_{qM}) \sim_K tr_{q,1}{:}q(s_{q1}, \cdots, s_{qM})$
and $tr_1 = \sigma_1'(\langle rID, p(\vec{x}_p), tr_{q,1}{:}q(x_{q1}, \cdots, x_{qM}), b_1(\vec{x}_{b1}) :: \cdots :: b_n(\vec{x}_{bn})\rangle) = \langle rID, p(\vec{t_p}), tr_{q,2}{:}q(t_{q1}, \cdots, t_{qM}), b_1(\vec{t}_{b1}) :: \cdots :: b_n(\vec{t}_{bn})\rangle$
we apply $\sim_K$-Base to obtain:
$tr_1{:}p(t_{p1}, \cdots, t_{pN}) \sim tr_2{:}p(s_{p1}, \cdots, s_{pN})$

Pick any $i \in [1, N]$.
Assume $t_i \neq s_i$.
Goal:
$\exists \ell \in [1, |e|]$ s.t.
$DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i,)$
and $\ell \notin K$

By definition of σ'_1 and σ'_2,
$\sigma'_1(p{:}i) \neq \sigma'_2(p{:}i)$

By definition of σ'_2, one of the following hold
(A) $\exists \ell \in [1, |e|]$ s.t. $DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i,)$ and $\ell \notin K$
otherwise if $\ell \in K$, then $\sigma'_1(e{:}\ell) = \sigma'_1(p{:}i) = t_i = s_i = \sigma'_2(p{:}i)\sigma'_2(e{:}\ell)$
(B): $\exists j \in [1, m]$ s.t. $p{:}i = y_j = \textsc{FunJ}(\vec{z}_j)$ and $\sigma'_1(\vec{z}_j \cap \vec{x}_e) \neq \sigma'_2(\vec{z}_j \cap \vec{x}_e)$

Case A $\exists \ell \in [1, |e|]$ *s.t.* $DQ \vdash \mathsf{connected}(e{:}\ell, p{:}i,)$ *and* $\ell \notin K$
The goal already holds

Case B: $\exists j \in [1, m]$ *s.t.* $p{:}i = y_j = \textsc{FunJ}(\vec{z}_j)$ *and* $\vec{z}_j \cap \vec{x}_e \neq \emptyset$ *and* $\sigma'_1(\vec{z}_j \cap \vec{x}_q) \neq \sigma'_2(\vec{z}_j \cap \vec{x}_q)$
By assumptions,
$\exists \ell \in [1, |q|]$ s.t. $q{:}\ell \in \vec{z}_j \cap \vec{x}_q$ and $\sigma'_1(q{:}\ell) \neq \sigma'_2(q{:}\ell)$
By the induction hypothesis,
$\exists k \in [1, |e|]$ s.t.
$DQ \vdash \mathsf{connected}(e{:}k, q{:}\ell)$ and $e{:}k \notin K$
Since $q{:}\ell \in \vec{z}_j$, by Join-Func-Attr we have:
$DQ \vdash \mathsf{joinSAttr}(q{:}\ell)$
By $DQ \vdash \mathsf{connected}(e{:}k, q{:}\ell)$ and $DQ \vdash \mathsf{joinSAttr}(q{:}\ell)$ and Equi-Reachable,
$e{:}k \in K$
This contradicts the the induction hypothesis that $e{:}k \notin K$

□

D. OPERATIONAL SEMANTICS OF SEMI-NAÏVE EVALUATION

The operational semantics of the semi-naïve evaluation of DELP programs adopts a distributed execution model. Each node runs a designated program, and maintains a database of proofs of derived tuples in its local state. Nodes can communicate with each other by sending tuples over the network. The evaluation of the DELP programs follows the PSN algorithm [10], and maintains the database incrementally.

At a high-level, each node computes its local fixed-point by firing the rules on newly derived tuples. The fixed-point computation can also be triggered when a node receives tuples from the network. When a tuple is derived, it is sent to the node specified by its location specifier. Instead of blindly computing the fixed-point, we make sure that only rules whose body tuples are updated are fired.

D.1 Hash functions

During our online compression execution, we hash the values of certain provenance elements in order to save on storage space or to generate unique identifiers. In order to show that semi-naïve evaluation is bisimular to online compression execution, we need to use some of the hash functions for online compression in semi-naïve evaluation as well. We present the algorithms used to compute these hash values in Figure D.1.

Given a declaration for the program DQ and an instance of its event relation $e(@\iota_e, \vec{c}_e)$, Algorithm EquiHash finds the equivalence keys K for e, returns equivalence hash value of $e(@\iota_e, \vec{c}_e)$.

Given a declaration for the program DQ and an instance of one of its relations $p(@\iota_p, \vec{c}_p)$, Algorithm TupleHash finds the primary keys *pkeys* for p, and returns the hash of $p(@\iota_p, \vec{c}_p)$ on its primary keys.

function EquiHash($e(@\iota_e, \vec{t}_e)$, Γ)
  $K \leftarrow \Gamma(e)[equi\_attr]$
  $i_1 :: \cdots :: i_n \leftarrow K$
  return hash($\vec{t}_e{:}i_1, \cdots, \vec{t}_e{:}i_n$)
end function

function TupleHash($p(@\iota_p, \vec{t}_p)$, Γ)
  $pkeys \leftarrow \Gamma(e)[primary\_keys]$
  $i_1 :: \cdots :: i_n \leftarrow pkeys$
  return hash($\vec{t}_p{:}i_1, \cdots, \vec{t}_p{:}i_n$)
end function

Figure 21: Hash functions used in program execution

D.2 Definitions of network states

In Figure D.2, we present the constructs needed for defining the operational semantics for Semi-naïve evaluation.

The network configuration $\mathcal{C}_{sn}$ for the entire system that runs the evaluation is represented as $\mathcal{Q}_{sn} \triangleright \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$. $\mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$ are the local network states for each node in the distributed system, while $\mathcal{Q}_{sn}$ is a queue of updates consisting of fast-changing tuples which will eventually be sent to the nodes specified by the location specifier.

Each node ι in the distributed system has local state $\mathcal{S}_{sn}$, where $\mathcal{S}_{sn} = \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{sn}, equiSet, \mathcal{M}, \mathcal{M}_{prov} \rangle$ consists of attributes needed to execute DQ locally.

The first four attributes of $\mathcal{S}_{sn}$ have been described in Appendixes B and C. We summarize them for completeness. We have (1) ι, the identifier of the local state, (2) DQ, the DELP program that is to be executed, (3) Γ, the mapping of every relation in DQ to a type, and (4) $\mathcal{DB}$, a local database of materialized tuples used to execute rules.

The new constructs in $\mathcal{S}_{sn}$ introduced are (5) $\mathcal{E}$, a set of instances of events in which e each element in $\mathcal{E}$ is an instance of the event relation triggering execution of DQ. Of particular importance is (6) $\mathcal{U}_{sn}$, a set of updates consisting of instance of fast-changing relations that trigger execution of rules in DQ. They differ from $\mathcal{Q}_{sn}$ as all updates in $\mathcal{U}_{sn}$ represent tuples which are locally stored, in contrast to $\mathcal{Q}_{sn}$ whose tuples can be stored anywhere in the network. Finally, we have (7) *equiSet*, a set of of hashes of all the equivalence keys that have been seen so far on node ι, (8) $\mathcal{M}$, a set of derivation trees of fast-changing tuples representing the provenance of rules fired during execution and finally (9) $\mathcal{M}_{prov}$, a set derivation trees of tuples that are instances of relations of interest.

Global network configuration	$\mathcal{C}_{sn}$	::=	$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$
Network queue	$\mathcal{Q}_{sn}$	::=	$\mathcal{U}_{sn}$
Update	u_{sn}	::=	$ev \mid tr{:}P$
Local state	$\mathcal{S}_{cm}$	::=	$\langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{sn}, equiSet, \mathcal{M}, \mathcal{M}_{prov} \rangle$
Event queue	$\mathcal{E}$	::=	$\mathsf{nil} \mid ev :: \mathcal{E}$
Local updates	$\mathcal{U}_{sn}$	::=	$[u_{sn_1}, \cdots, u_{sn_n}]$
Equivalence hash table	$equiSet$	::=	$\cdot \mid equiSet, heq$
Rule provenance table	$\mathcal{M}$	::=	$\cdot \mid \mathcal{M}, tr{:}P$
Tuple provenance table	$\mathcal{M}_{prov}$	::=	$\cdot \mid \mathcal{M}_{prov}, prov$
Tuple provenance	$prov$	::=	$\mathsf{interest}(tr)$

Figure 22: Definitions of network state for Semi-naïve evaluation

D.3 Evaluation rules

We introduce the transition rules and explain how configurations are updated based on the updates in the network queue.

Global state transition ($\mathcal{C}_{sn} \to \mathcal{C}_{sn}'$).

The small-step operational semantics of the entire distributed system is denoted $\mathcal{C}_{sn} \to^n \mathcal{C}_{sn}'$, where n is the number of steps taken to transition from the initial state $\mathcal{C}_{sn_{init}}$ to $\mathcal{C}_{sn}'$. A trace $\mathcal{T}$ is a sequence of transitions $\mathcal{C}_{sn_{init}} \to^0 \mathcal{C}_{sn_1} \to^1 \cdots \to^n \mathcal{C}_{sn_{n+1}}$.

Rule SN-NodeStep states that the system takes a step when one node takes a step. As a result, the updates generated by node ι are appended to the end of the network queue. We use $\circ$ to denote the list append operation. Rule SN-DeQueue applies when a node receives updates from the network. We write $\mathcal{E}_1 \oplus \mathcal{E}_2$ to denote a merge of two lists. Any node can dequeue updates sent to it and append those updates to the update list in its local state. Here, we overload the $\circ$ operator, and write $\mathcal{Q}_{sn} \circ \mathcal{E}$ to denote a new state, which is the same as $\mathcal{Q}_{sn}$, except that the update list is the result of appending $\mathcal{E}$ to the update list in $\mathcal{Q}_{sn}$.

Local state transition ($\mathcal{S}_{sn} \hookrightarrow \mathcal{S}_{sn}', \mathcal{U}_{sn}$).

From state $\mathcal{S}_{sn}$, a node takes a step to a new state $\mathcal{S}_{sn}'$ and generates a set of updates $\mathcal{U}_{sn}$ for other nodes in the network. This is denote by $\mathcal{S}_{sn} \hookrightarrow \mathcal{S}_{sn}', \mathcal{U}_{sn}$.

Each program DQ is triggered by instance of the event relation e. Each node ι_e contains a queue $\mathcal{E}$ of instances of e. Rule SN-Event states that an execution of DQ is triggered by dequeuing an element $e(@\iota_e, \vec{c}_e)$ in $\mathcal{E}$ and placing it into the set of local updates $\mathcal{U}_{sn}$.

Each u_{sn} in the set of local updates $\mathcal{U}_{sn}$ on node ι_q denotes a derivation tree of a fast-changing tuple $q(@\iota_q, \vec{c}_q)$. $q(@\iota_q, \vec{c}_q)$ can be used to trigger more rules in DQ. *fireRulesSN* takes in arguments ι_q, ΔDQ, u_{sn}, $\mathcal{DB}$, and $\mathcal{M}$, and fires all rules in DQ that are triggered when given u_{sn} and $\mathcal{DB}$. It then returns a set of local updates $\mathcal{U}_{sn_{in}}$, a set of external updates $\mathcal{U}_{sn_{ext}}$ consisting of tuples that are to be sent to other nodes in the distributed system, and the set of updated derivation trees of tuples $\mathcal{M}'$ that represent the provenance of the rules that have been fired locally.

Fire Rules ($fireRulesSN(@\iota, \Delta \boldsymbol{DQ}, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}, \mathcal{U}_{sn_{ext}}, \mathcal{M}')$).

Given one update, we fire rules in program DQ that are affected by this update.

Rule SN-Empty is the base case where all rules have been fired, so we directly return empty update sets and the same set of derivation trees of tuples generated.

Given a program $[\Delta r, \Delta DQ']$ (where DQ' can be the empty list) with at least one rule, rule SN-Seq first fires the rule Δr, then recursively calls itself to process the rest of the rules in $\Delta DQ'$. The resulting updates and derivation trees generated are the union of the updates and derivation trees generated by firing Δr and $\Delta DQ'$.

Fire a single rule ($fireSingleRuleSN(@\iota, \Delta r, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}')$).

Given one update, one rule, and a database of materialized slow-changing tuples, we find all possible substitutions Σ that satisfy the rule body. We may choose to only fire rules using a subset of all possible substitutions For each possible substitution we want to use, we find the sets of updates and derivation trees generated by firing the rule.

Fire a single rule given substitutions ($derivationSN(@\iota, \Sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}')$).

Given one update, one rule, and a list of substitutions for relations in body of the rule, we derive the head of the rule.

Rule SN-Subst-Empty is the base case when there are no more substitutions, so we directly return empty update sets and the same set of and derivation trees of tuples generated.

Given that there is at least one substitution $\sigma :: \Sigma$, rule SN-Subst first derives the update triggered by σ, then recursively calls itself to process the rest of the substitutions in $\Delta\Sigma$. The resulting updates and derivation trees generated are the union of the updates and derivation trees generated by Δr and $\Delta\Sigma$.

Fire a single rule given one substitution ($singleDerivSN(@\iota, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}')$).

Given a substitution σ for the rule body of rule Δr, rule SN-SingleSubst derives the head tuple of Δr. If the head is also located at node ι, the head tuple is an internal update. Otherwise, the head tuple is an external update. We update the set of and derivation trees of tuples derived locally to include the and derivation tree for the head tuple.

$\boxed{\mathcal{C}_{sn} \to \mathcal{C}_{sn}'}$

$$\frac{\mathcal{S}_{sn_i} \hookrightarrow \mathcal{S}_{sn_i}', \mathcal{U}_{sn} \qquad \forall j \in [1,N] \wedge j \neq i, \mathcal{S}_{sn_j}' = \mathcal{S}_{sn_j}}{\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N} \rightarrow \mathcal{Q}_{sn} \circ \mathcal{U}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}} \text{ SN-NodeStep}$$

$$\frac{\mathcal{Q}_{sn} = \mathcal{Q}_{sn}{}' \oplus \mathcal{Q}_{sn_1} \oplus \cdots \oplus \mathcal{Q}_{sn_N}}{\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N} \rightarrow \mathcal{Q}_{sn}{}' \rhd (\mathcal{S}_{sn_1} \circ \mathcal{Q}_{sn_1}) \cdots (\mathcal{S}_{sn_N} \circ \mathcal{Q}_{sn_N})} \text{ SN-DeQueue}$$

$\boxed{\mathcal{S}_{sn} \hookrightarrow \mathcal{S}_{sn}', \mathcal{U}_{sn}}$

$$\frac{\begin{array}{c} u_{sn} = e(@\iota_e, \vec{t}_e) \\ \Gamma(e)[tuple] = \mathsf{event} \qquad K = \Gamma(e)[equi\_attr] \qquad heq = \textsc{EquiHash}(e(@\iota_e, \vec{t}_e), K) \qquad equiSet' = equiSet \cup heq \end{array}}{\langle @\iota, DQ, \Gamma, \mathcal{DB}, e(@\iota_e, \vec{t}_e) :: \mathcal{E}, \mathcal{U}_{sn}, equiSet, \mathcal{M}, \mathcal{M}_{prov} \rangle \hookrightarrow \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{sn} \circ [u_{sn}], equiSet', \mathcal{M}, \mathcal{M}_{prov} \rangle} \text{ SN-Event}$$

$$\frac{\Gamma(q)[tuple] = \mathsf{fast} \qquad u_{sn} = tr_q{:}q(@\iota_q, \vec{t}_q) \qquad fireRulesSN(@\iota_q, \Delta DQ, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}')}{\begin{array}{c} \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{sn} :: \mathcal{U}_{sn}, \mathcal{M}, \mathcal{M}_{prov} \rangle \\ \hookrightarrow \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{sn} \circ \mathcal{U}_{sn_{in}}', equiSet, \mathcal{M}', \mathcal{M}_{prov} \rangle, \mathcal{U}_{sn_{ext}}' \end{array}} \text{ SN-RuleFire-Fast}$$

$$\frac{\Gamma(q)[tuple] = \mathsf{interest} \qquad u_{sn} = tr_q{:}q(@\iota_q, \vec{t}_q)}{\begin{array}{c} \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{sn} :: \mathcal{U}_{sn}, equiSet, \mathcal{M}, \mathcal{M}_{prov} \rangle \\ \hookrightarrow \langle @\iota, DQ, \Gamma, \mathcal{DB} \cup q(@\iota_q, \vec{t}_q), \mathcal{E}, \mathcal{U}_{sn} \circ \mathcal{U}_{sn_{in}}', equiSet, \mathcal{M}', \mathcal{M}_{prov} \cup \mathsf{interest}(tr_q{:}q(@\iota_q, \vec{t}_q)) \rangle, \mathcal{U}_{sn_{ext}}' \end{array}} \text{ SN-RuleFire-Interest}$$

$\boxed{fireRulesSN(@\iota, \Delta DQ, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}, \mathcal{U}_{sn_{ext}}, \mathcal{M}')}$

$$\frac{}{fireRulesSN(@\iota, [], u_{sn}, \mathcal{DB}, \mathcal{M}) = ([], [], \mathcal{M})} \text{ SN-Empty}$$

$$\frac{\begin{array}{c} fireSingleRuleSN(@\iota, \Delta r, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}') \\ fireRulesSN(@\iota, \Delta DQ, u_{sn}, \mathcal{DB}, \mathcal{M}') = (\mathcal{U}_{sn_{in}}'', \mathcal{U}_{sn_{ext}}'', \mathcal{M}'') \end{array}}{fireRulesSN(@\iota, (\Delta r, \Delta DQ), u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}' \circ \mathcal{U}_{sn_{in}}'', \mathcal{U}_{sn_{ext}}' \circ \mathcal{U}_{sn_{ext}}'', \mathcal{M}'')} \text{ SN-Seq}$$

$\boxed{fireSingleRuleSN(@\iota, \Delta r, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}')}$

$$\frac{\begin{array}{c} \Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \qquad u_{sn} = tr_q{:}q(@\iota_q, \vec{t}_q) \\ \Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}) \qquad \Sigma' = sel(\Sigma, \Delta r) \qquad derivationSN(@\iota_q, \Sigma', \Delta r, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}') \end{array}}{fireSingleRuleSN(@\iota_q, \Delta r, u_{sn}, \mathcal{DB}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}')} \text{ SN-FireSingle}$$

$\boxed{derivationSN(@\iota, \Sigma, \Delta r, u_{sn}, \mathcal{M} = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}')}$

$$\frac{}{derivationSN(@\iota, [], \Delta r, u_{sn}, \mathcal{M}) = ([], [], \mathcal{M})} \text{ SN-Subst-Empty}$$

$$\frac{singleDerivSN(@\iota, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}') \qquad derivationSN(@\iota, \Sigma, \Delta r, u_{sn}, \mathcal{M}') = (\mathcal{U}_{sn_{in}}'', \mathcal{U}_{sn_{ext}}'', \mathcal{M}'')}{derivationSN(@\iota, \sigma :: \Sigma, \Delta r, u_{sn}, \mathcal{M}'') = (\mathcal{U}_{sn_{in}}' \circ \mathcal{U}_{sn_{in}}'', \mathcal{U}_{sn_{ext}}' \circ \mathcal{U}_{sn_{ext}}'', \mathcal{M}'')} \text{ SN-Subst}$$

$\boxed{singleDerivSN(@\iota, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}')}$

$$\frac{\begin{array}{c}\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta e(@\ell_e, \vec{x}_e), b_1(@\ell_e, \vec{x}_{b1}), \cdots, b_n(@\ell_e, \vec{x}_{bn}), \cdots \qquad u_{sn} = e(@\iota_e, \vec{t}_e) \\ e(@\ell_e, \vec{x}_e)\sigma = e(@\iota_e, \vec{t}_e) \qquad \Gamma(e)[type] = \mathsf{event} \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_e \cup \vec{x}_e \cup \bigcup_{i=1}^{N} \vec{x}_{bi} \\ tr_p = (rID, p(@\ell_p, \vec{x}_p)\sigma, e(@\iota_e, \vec{t}_e), b_1(@\ell_e, \vec{x}_{b1})\sigma :: \cdots :: b_n(@\ell_e, \vec{x}_{bn})\sigma) \qquad u_{sn}{}' = tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \\ \text{if } \sigma(@\ell_p) = @\iota_e \text{ then } \mathcal{U}_{sn}{}'_{in} = [u_{sn}{}'], \mathcal{U}_{sn}{}'_{ext} = [] \text{ else } \mathcal{U}_{sn}{}'_{in} = [], \mathcal{U}_{sn}{}'_{ext} = [u_{sn}{}'] \\ \mathcal{M}' = \mathcal{M} \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \end{array}}{singleDerivSN(@\iota_q, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn}{}'_{in}, \mathcal{U}_{sn}{}'_{ext}, \mathcal{M})}\ \text{SN-SingleSubst-Event}$$

$$\frac{\begin{array}{c}\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \\ u_{sn} = tr_q{:}q(@\iota_q, \vec{t}_q) \qquad \text{either } \Gamma(q)[type] = \mathsf{fast} \text{ or } \Gamma(q)[type] = \mathsf{interest} \\ q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{N} \vec{x}_{bi} \\ tr_p = (rID, p(@\ell_p, \vec{x}_p)\sigma, tr_q{:}q(@\iota_q, \vec{t}_q), b_1(@\ell_q, \vec{x}_{b1}))\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma) \qquad u_{sn}{}' = tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \\ \text{if } \sigma(@\ell_p) = @\iota_q \text{ then } \mathcal{U}_{sn}{}'_{in} = [u_{sn}{}'], \mathcal{U}_{sn}{}'_{ext} = [] \text{ else } \mathcal{U}_{sn}{}'_{in} = [], \mathcal{U}_{sn}{}'_{ext} = [u_{sn}{}'] \\ \mathcal{M}' = \mathcal{M} \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \end{array}}{singleDerivSN(@\iota_q, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn}{}'_{in}, \mathcal{U}_{sn}{}'_{ext}, \mathcal{M}')}\ \text{SN-SingleSubst-Fast}$$

E. OPERATIONAL SEMANTICS OF ONLINE COMPRESSION EXECUTION

Our online compression scheme compresses equivalent distributed provenance trees based on equivalence keys identified. We store one representative provenance tree for all provenances in the same equivalence class.

The operational semantics of the online compression evaluation for DELP programs are similar to the operation semantics for semi-naïve evaluation introduced in Appendix D. However, some of the constructs used to define the set of updates and proofs generated are different. Appendix D motivates and describes these differences.

Next, we present an evaluation strategy that shares the storage of provenances *within* the same equivalence class in Appendix E.1. Building on this, we next present an evaluation strategy that allows for bigger gains in storage space saved by sharing the storage of provenances *across* equivalence classes in Appendix E.2.

E.1 Sharing storage *within* equivalence classes

In this section, we describe an evaluation strategy to shares the storage of provenances *within* the same equivalence class. We store the provenance of each rule fired in the form of a provenance node with a reference to the provenance of the previous rule fired in order to recover the complete provenance of a tuple during provenance querying.

E.1.1 Definitions of network states

All the constructs used to represent the online compression evaluation have analogous functions to their respective counterparts in semi-naïve evaluation. Many of the constructs are identical to those of semi-naïve evaluation. However, the constructs that deal with provenance storage are different, as online compression saves storage space by only storing the provenance of the rule that derived tuple P instead of storing the entire derivation tree of a tuple derived locally in semi-naïve evaluation.

The network configuration $\mathcal{C}_{cm}$ for online compression execution is represented as $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$. Similar to the network configuration $\mathcal{C}_{sn}$ (where $\mathcal{C}_{sn} = \mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$)for semi-naïve evaluation, $\mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$ are the local network states for each node in the distributed system, while $\mathcal{Q}_{cm}$ is a queue of updates consisting of fast-changing tuples which will eventually be sent to the nodes specified by the location specifier.

Each node ι in the network has local state $\mathcal{S}_{cm}$, where $\mathcal{S}_{cm} = \langle \iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov} \rangle$. Most of the attributes in $\mathcal{S}_{cm}$ are identical to their counterparts in $\mathcal{S}_{sn}$. We summarize the differing constructs of each local state in Figure E.1.1. In particular, the set of local updates $\mathcal{U}_{cm}$, the set of local provenances Υ, and the set of tuple provenances representing relations of interest Υ_{prov} differ from those of semi-naïve evaluation.

Global Network Configuration	$\mathcal{C}_{cm}$	::=	$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
Network Queue	$\mathcal{Q}_{cm}$	::=	$\mathcal{U}_{cm}$
Local State	$\mathcal{S}_{cm}$	::=	$\langle \iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov} \rangle$
Updates	$\mathcal{U}_{cm}$	::=	$\{u_{cm1}, \cdots, u_{cmn}\}$
Collection of rule provenances	Υ	::=	$\cdot \mid \Upsilon, ruleExec$
Collection of tuple provenances	Υ_{prov}	::=	$\cdot \mid \Upsilon_{prov}, prov$

Figure 23: Definitions of network state for online compression evaluation

Rule provenances are stored differently because online compression saves storage by recording provenance information more concisely than semi-naïve evaluation does. Figure E.1.1 summarizes the constructs use by online compression to record rule provenances.

Instead of recording the entire tuple, online compression records only the hash of the primary keys of a tuple. We write `eID`, `vID`, and `tID` to refer to the hash of the primary keys of an event tuple, slow-changing tuple, and tuple of a relation of interest respectively. Instead of recording the entire provenance tree for each new fast-changing tuple derived during program execution, online compression records only the provenance of the new rule fired as *ruleargs* on the node at which the rule was fired. Thus, the provenance elements representing the derivation of a single tuple may be stored on several different nodes in the network. Because different executions may use the same arguments to fire a particular rule, each rule provenance element *ruleExec* records a lookup key λ unique to it. It also records the lookup key of the previous tuple that trigger the rule. We denote an ordered list of rule provenance elements representing the provenance of a tuple as *yl*.

Unique ID for an event tuple	`eID`	::=	$\textsc{EquiHash}(ev, \Gamma)$
Unique ID for a slow-changing tuple	`vID`	::=	$\textsc{TupleHash}(P, \Gamma)$
Unique ID for a tuple of a relation of interest	`tID`	::=	$\textsc{TupleHash}(res, \Gamma)$
Provenance for a single rule execution	*ruleargs*	::=	$rID :: \iota :: \texttt{vID}_1 :: \cdots :: \texttt{vID}_n$
Hash of ruleargs	`HrID`	::=	$\mathsf{hash}(ruleargs)$
Unique identifier for a rule provenance element	$\flat$	::=	$\mathsf{hash}(\lambda)$
Lookup Keys	λ	::=	$\mathsf{id}(\emptyset, \emptyset, heq) \mid \mathsf{id}(@\iota, \texttt{HrID}, \flat)$
Rule provenance element	*ruleExec*	::=	$\langle \lambda_p, ruleargs, \lambda_q \rangle$
Provenance for a DELP program execution	*yl*	::=	$\mathsf{nil} \mid yl :: ruleExec$

Figure 24: Constructs used to record rule provenances during online compression execution

An update u_{cm} in online compression evaluation differs from is counterpart u_{sn} in semi-naïve evaluation. u_{sn} is the entire provenance tree of a tuple P. In contrast, the corresponding update u_{cm} for tuple P does not store the complete provenance tree for P to save bandwidth. Instead, u_{cm} has form $\langle P, \mathsf{createFlag}, \texttt{eID}, \lambda \rangle$, in which createFlag is a flag that identifies whether provenances should be created and maintained during program execution, `eID` is the hash of the event tuple that triggered program execution, and λ represents the lookup key that enables us the retrieve the rule provenance that derived tuple P.

Create Flag	createFlag	::=	*Create* | *NCreate*
Update	u_{cm}	::=	$\langle P, \mathsf{createFlag}, \texttt{eID}, \lambda \rangle$

Figure 25: Definition of updates for online compression with sharing *within* equivalence class

E.1.2 Evaluation rules

Most of the transition rules are similar to those in appendix D.3. The transition rules that maintain provenance ($singleDerivSN(@\iota, \sigma, \Delta r, u_{sn}, \mathcal{M}) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}')$ for semi-naïve evaluation and $singleCompressionCM(@\iota, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')$ for online compression evaluation) are different. We explain the rules that differ below.

Fire single rule given one substitution ($singleCompressionCM(@\iota, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')$)

If the update consists of a tuple and a flag instructing us to maintain provenance, we execute Rule CM-Create and generate a new update consisting of the head of rule r, and adds the rule provenance for this execution of r to the set of local rule provenances.

Otherwise, if the update consists of a tuple and a flag instructing us *not* to maintain provenance, we execute rule CM-NCreate to generate a new update consisting of the head of rule r.

$\boxed{\mathcal{C}_{cm} \rightarrow \mathcal{C}_{cm}{}'}$

$$\frac{\mathcal{S}_{cm_i} \hookrightarrow \mathcal{S}_{cm'_i}, \mathcal{U}_{cm} \qquad \forall j \in [1,n] \wedge j \neq i, \mathcal{S}_{cm'_j} = \mathcal{S}_{cm_j}}{\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_n} \rightarrow \mathcal{Q}_{cm} \circ \mathcal{U}_{cm} \rhd \mathcal{S}_{cm'_1} \cdots \mathcal{S}_{cm'_n}} \text{ CM-NodeStep}$$

$$\frac{\mathcal{Q}_{cm} = \mathcal{Q}_{cm}{}' \oplus \mathcal{Q}_{cm_1} \oplus \cdots \oplus \mathcal{Q}_{cm_n}}{\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_n} \rightarrow \mathcal{Q}_{cm}{}' \rhd (\mathcal{S}_{cm_1} \circ \mathcal{Q}_{cm_1}) \cdots (\mathcal{S}_{cm_n} \circ \mathcal{Q}_{cm_n})} \text{ CM-DeQueue}$$

$\boxed{\mathcal{S}_{cm} \hookrightarrow \mathcal{S}_{cm}', \mathcal{U}_{cm}}$

$$\frac{\Gamma(e)[tuple] = \mathsf{event} \qquad \mathtt{eID} = \textsc{TupleHash}(e(@\iota_e, \vec{t_e}), \Gamma) \\ heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma) \qquad \text{If } heq \in equiSet \text{ then } \mathsf{createFlag} = NCreate \text{ else } \mathsf{createFlag} = Create \\ ucm = \langle e(@\iota_e, \vec{t_e}), \mathsf{createFlag}, \mathtt{eID}, \mathsf{id}(\emptyset, \emptyset, heq)\rangle \qquad equiSet' = equiSet \cup heq}{\langle @\iota_e, DQ, \Gamma, \mathcal{DB}, ev :: \mathcal{E}, \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov}\rangle \hookrightarrow \langle @\iota_e, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm} \circ [u_{cm}], equiSet', \Upsilon, \Upsilon_{prov}\rangle, []} \text{ CM-Init-Event}$$

$$\frac{ucm = \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle \qquad \text{either } \Gamma(q)[tuple] = \mathsf{fast} \text{ or } \Gamma(q)[tuple] = \mathsf{interest} \\ fireRulesCM(@\iota_q, \Delta DQ, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')}{\langle @\iota_q, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{cm} :: \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov}\rangle \hookrightarrow \langle @\iota_q, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm} \circ \mathcal{U}_{cm'_{in}}, equiSet, \Upsilon', \Upsilon_{prov}\rangle, \mathcal{U}_{cm_{ext}}} \text{ CM-RuleFire-Intm}$$

$$\frac{ucm = \langle p(@\iota_p, \vec{t_p}), \mathsf{createFlag}, \mathtt{eID}, \lambda_p\rangle \qquad \Gamma(p)[tuple] = \mathsf{interest} \\ \mathtt{tID} = \textsc{TupleHash}(p(@\iota_p, \vec{t_p}), \Gamma) \qquad prov = \langle @\iota_p, \mathtt{tID}, \mathtt{eID}, \lambda_p\rangle \qquad \Upsilon_{prov'} = \Upsilon_{prov} \cup prov}{\langle @\iota_p, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{cm} :: \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov}\rangle \\ \hookrightarrow \langle @\iota_p, DQ, \Gamma, \mathcal{DB} \cup \{p(@\iota_p, \vec{t_p})\}, \mathcal{E}, \mathcal{U}_{cm}, equiSet, \Upsilon, \Upsilon_{prov'}\rangle, []} \text{ CM-RuleFire-Interest}$$

$\boxed{fireRulesCM(@\iota, \Delta DQ, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \Upsilon')}$

$$\frac{}{fireRulesCM(@\iota, [], u_{cm}, \mathcal{DB}, \Upsilon) = ([], [], \Upsilon)} \text{ CM-Empty}$$

$$\frac{fireSingleRuleCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \Upsilon') \qquad fireRulesCM(@\iota, \Delta DQ, u_{cm}, \mathcal{DB}, \Upsilon') = (\mathcal{U}''_{in}, \mathcal{U}''_{ext}, \Upsilon'')}{fireRulesCM(@\iota, \Delta r :: \Delta DQ, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}'_{in} \circ \mathcal{U}''_{in}, \mathcal{U}'_{ext} \circ \mathcal{U}''_{ext}, \Upsilon'')} \text{ CM-Seq}$$

$\boxed{fireSingleRuleCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \Upsilon')}$

$$\frac{\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), \cdots \qquad ucm = \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle \\ \Sigma = \rho(\Delta r, q(@\iota_q, \vec{t_q}), \mathcal{DB}) \qquad \Sigma' = sel(\Sigma, \Delta r) \qquad compressionCM(@\iota_q, \Sigma', \Delta r, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')}{fireSingleRuleCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \Upsilon) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \Upsilon')} \text{ CM-FireSingle}$$

$\boxed{compressionCM(@\iota, \Sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')}$

$$\frac{}{compressionCM(@\iota, [], \Delta r, u_{cm}, \Upsilon) = ([], [], \Upsilon)} \text{ CM-Compress-Empty}$$

$$\frac{singleCompressionCM(@\iota, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon') \\ compressionCM(@\iota, \Sigma, \Delta r, u_{cm}, \Upsilon') = (\mathcal{U}_{cm''_{in}}, \mathcal{U}_{cm''_{ext}}, \Upsilon'')}{compressionCM(@\iota, \sigma :: \Sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}} \circ \mathcal{U}_{cm''_{in}}, \mathcal{U}_{cm'_{in}} \circ \mathcal{U}_{cm''_{ext}}, \Upsilon'')} \text{ CM-Compress}$$

$\boxed{singleCompressionCM(@\iota, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')}$

$$\frac{\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \\ ucm = \langle q(@\iota_q, \vec{t_q}), Create, \mathtt{eID}, \lambda_q\rangle \qquad q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t_q}) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi} \\ \forall i \in [1, n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma) \qquad ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \\ \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \text{if } (\Gamma(q)[type] = \mathsf{event}) \text{ then } \flat_p = \mathsf{hash}(\lambda_q{:}3) \text{ else } \flat_p = \mathsf{hash}(\lambda_q) \\ \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p) \qquad ucm' = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p\rangle \qquad ruleExec_p = \langle \lambda_p, ruleargs, \lambda_q\rangle \\ \Upsilon' = \Upsilon \cup ruleExec_p \qquad \text{if } \sigma(@\ell_p) = @\iota_q \text{ then } \mathcal{U}_{cm'_{in}} = [ucm'], \mathcal{U}_{cm'_{ext}} = [] \text{ else } \mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [ucm']}{singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon')} \text{ CM-Create}$$

$$\frac{\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \\ ucm = \langle q(@\iota_q, \vec{t_q}), NCreate, \mathtt{eID}, \lambda_q\rangle \qquad q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t_q}) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi} \\ \forall i \in [1, n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma) \qquad ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \\ \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \text{if } (\Gamma(q)[type] = \mathsf{event}) \text{ then } \flat_p = \mathsf{hash}(\lambda_q{:}3) \text{ else } \flat_p = \mathsf{hash}(\lambda_q) \\ \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p) \qquad ucm' = \langle p(@\ell_p, \vec{x}_p)\sigma, NCreate, \mathtt{eID}, \lambda_p\rangle \\ \text{if } \sigma(@\ell_p) = @\iota_q \text{ then } \mathcal{U}_{cm'_{in}} = [ucm'], \mathcal{U}_{cm'_{ext}} = [] \text{ else } \mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [ucm']}{singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm}, \Upsilon) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon)} \text{ CM-NCreate}$$

E.2 Sharing storage *across* equivalence classes

In this section, we describe an evaluation strategy to shares the storage of provenances *across* equivalence classes. Instead of storing the provenance of each rule fired together with the reference to the provenance of the previous rule fired, we store these two pieces of information separately. While each provenance node recording the parent-child relationship between rules fired is unique to a particular equivalence class, the provenance element recording the provenance of each rule fired can be shared between different equivalence classes.

E.2.1 Definitions of network states

The operational semantics of the online compression evaluation with sharing *across* equivalence class are similar to the operational semantics of the online compression evaluation *without* sharing across equivalence class, as described in appendix E.1.2.

However, the constructs used to store the provenances generated are necessarily different. We summarize the differing constructs in Figure E.2.1. When sharing provenance *within* equivalence classes, we store both the arguments used to fire a DELP rule and parent-child relationship between the rule provenance representing the previous rule fired together as *ruleExec*. Each *ruleExec* has form $\langle \lambda_p, ruleargs_p, \lambda_q \rangle$, in which λ_p is the lookup key is for the current rule provenance, $ruleargs_p$, and λ_q is the lookup key for the previous rule provenance. In contrast, when sharing provenance *across* equivalence classes, we store the parent-child relationship between rule provenances separately. We store the arguments used for a single rule execution as a node ncm (where $ncm = (\langle @\iota, \texttt{HrID} \rangle, ruleargs)$), and the parent-child relationship between ncm and the previous rule execution as $lcm = (\langle @\iota, \texttt{HrID}, \flat \rangle, \lambda_q)$, where $\langle @\iota, \texttt{HrID}, \flat \rangle$ is an extension of the lookup id for ncm and λ_q is an extension of the lookup id for the provenance of the previous rule fired.

Global Network Configuration	$\mathcal{C}_{tcm}$	::=	$\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
Local State	$\mathcal{T}_{cm}$	::=	$\langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm}, equiSet, \mathcal{N}, \mathcal{L}, \Upsilon_{prov} \rangle$
Parent-child relationship between rule provenances	lcm	::=	(λ_p, λ_q)
Collection of parent-child relationships	$\mathcal{L}$	::=	$\cdot \mid \mathcal{L}, lcm$
Ordered list of rule provenances	ch	::=	$\mathsf{nil} \mid ch \rightsquigarrow (lcm :: ncm)$
Rule provenance	ncm	::=	$(\langle @\iota, \texttt{HrID} \rangle, ruleargs)$
Collection of rule provenances	$\mathcal{N}$	::=	$\cdot \mid \mathcal{N}, ncm$

Figure 26: Definition of network states for online compression with sharing *across* equivalence class

E.2.2 Evaluation rules

Most of the transition rules are similar to those in appendix E.1.2. The transition rules that handle provenance maintenance for online compression evaluation that shares storage within an equivalence class and online compression evaluation that shares storage across equivalence classes are necessarily different. We explain the rules that differ below.

Fire a single rule given one substitution ($singleCompressionAcrossCM(@\iota, \sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')$**)**

If the update consists of a tuple and a flag instructing us to maintain provenance, Rule CM-across-Create generates a new update consisting of the head of rule r, and adds the node for the rule provenance for this execution of r and the relationship between this rule provenance node for the new update and the rule provenance node for the tuple that triggered the update.

Otherwise, if the update consists of a tuple and a flag instructing us *not* to maintain provenance, Rule CM-across-NCreate generates a new update consisting of the head of rule r.

$\boxed{\mathcal{C}_{tcm} \rightarrow \mathcal{C}_{tcm'}}$

$$\frac{\mathcal{T}_{cm_i} \hookrightarrow \mathcal{T}_{cm'_i}, \mathcal{U}_{cm} \qquad \forall j \in [1,n] \wedge j \neq i, \mathcal{T}_{cm'_j} = \mathcal{T}_{cm_j}}{\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_n} \rightarrow \mathcal{Q}_{cm} \circ \mathcal{U}_{cm} \rhd \mathcal{T}_{cm'_1} \cdots \mathcal{T}_{cm'_n}} \text{ CM-across-NodeStep}$$

$$\frac{\mathcal{Q}_{cm} = \mathcal{Q}_{cm'} \oplus \mathcal{Q}_{cm_1} \oplus \cdots \oplus \mathcal{Q}_{cm_n}}{\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_n} \rightarrow \mathcal{Q}_{cm'} \rhd (\mathcal{T}_{cm_1} \circ \mathcal{Q}_{cm_1}) \cdots (\mathcal{T}_{cm_n} \circ \mathcal{Q}_{cm_n})} \text{ CM-across-DeQueue}$$

$\boxed{\mathcal{T}_{cm} \hookrightarrow \mathcal{T}_{cm'}, \mathcal{U}_{cm}}$

$$\frac{\Gamma(e)[tuple] = \mathsf{event} \qquad \mathtt{eID} = \textsc{TupleHash}(e(@\iota_e, \vec{t_e}), \Gamma) \qquad heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma) \\ \text{If } heq \in equiSet \text{ then } \mathsf{createFlag} = NCreate \text{ else } \mathsf{createFlag} = Create \\ u_{cm} = \langle e(@\iota_e, \vec{t_e}), \mathsf{createFlag}, \mathtt{eID}, \mathsf{id}(\emptyset, \emptyset, heq)\rangle \qquad equiSet' = equiSet \cup heq}{\begin{array}{c}\langle @\iota_e, DQ, \Gamma, \mathcal{DB}, ev :: \mathcal{E}, \mathcal{U}_{cm}, equiSet, \mathcal{N}, \mathcal{L}, \Upsilon_{prov}\rangle \\ \hookrightarrow \langle @\iota_e, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm} \circ [u_{cm}], equiSet', \mathcal{N}, \mathcal{L}, \Upsilon_{prov}\rangle, []\end{array}} \text{ CM-across-Init-Event}$$

$$\frac{u_{cm} = \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle \qquad \text{either } \Gamma(q)[tuple] = \mathsf{fast} \text{ or } \Gamma(q)[tuple] = \mathsf{interest} \\ fireRulesCM(@\iota_q, \Delta DQ, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')}{\begin{array}{c}\langle @\iota_q, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{cm} :: \mathcal{U}_{cm}, equiSet, \mathcal{N}, \mathcal{L}, \Upsilon_{prov}\rangle \\ \hookrightarrow \langle @\iota_q, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}_{cm} \circ \mathcal{U}_{cm'_{in}}, equiSet, \mathcal{N}', \mathcal{L}', \Upsilon_{prov}\rangle, \mathcal{U}_{cm_{ext}}\end{array}} \text{ CM-across-RuleFire-Intm}$$

$$\frac{u_{cm} = \langle p(@\iota_p, \vec{t_p}), \mathsf{createFlag}, \mathtt{eID}, \lambda_p\rangle \qquad \Gamma(p)[tuple] = \mathsf{interest} \\ \mathtt{tID} = \textsc{TupleHash}(p(@\iota_p, \vec{t_p}), \Gamma) \qquad prov = \langle @\iota_p, \mathtt{tID}, \mathtt{eID}, \lambda_p\rangle \qquad \Upsilon_{prov'} = \Upsilon_{prov} \cup prov}{\begin{array}{c}\langle @\iota_p, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, u_{cm} :: \mathcal{U}_{cm}, equiSet, \mathcal{N}, \mathcal{L}, \Upsilon_{prov}\rangle \\ \hookrightarrow \langle @\iota_p, DQ, \Gamma, \mathcal{DB} \cup p(@\iota_p, \vec{t_p}), \mathcal{E}, \mathcal{U}_{cm}, equiSet, \mathcal{N}, \mathcal{L}, \Upsilon_{prov'}\rangle, []\end{array}} \text{ CM-across-RuleFire-Interest}$$

$$\boxed{fireRulesAcrossCM(@\iota, \Delta DQ, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \mathcal{N}', \mathcal{L}')}$$

$$\frac{}{fireRulesAcrossCM(@\iota, [], u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = ([], [], \mathcal{N}, \mathcal{L})} \text{ CM-across-Empty}$$

$$\frac{fireSingleRuleAcrossCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \mathcal{N}', \mathcal{L}') \\ fireRulesAcrossCM(@\iota, \Delta DQ, u_{cm}, \mathcal{DB}, \mathcal{N}', \mathcal{L}') = (\mathcal{U}''_{in}, \mathcal{U}''_{ext}, \mathcal{N}'', \mathcal{L}'')}{fireRulesAcrossCM(@\iota, \Delta r :: \Delta DQ, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}'_{in} \circ \mathcal{U}''_{in}, \mathcal{U}'_{ext} \circ \mathcal{U}''_{ext}, \mathcal{N}'', \mathcal{L}'')} \text{ CM-across-Seq}$$

$$\boxed{fireSingleRuleAcrossCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \mathcal{N}', \mathcal{L}')}$$

$$\frac{\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \coloneq \Delta q(@\ell_q, \vec{x}_q), \cdots \\ u_{cm} = \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle \qquad \Sigma = \rho(\Delta r, q(@\iota_q, \vec{t_q}), \mathcal{DB}) \\ \Sigma' = sel(\Sigma, \Delta r) \qquad compressionAcrossCM(@\iota_q, \Sigma', \Delta r, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')}{fireSingleRuleAcrossCM(@\iota, \Delta r, u_{cm}, \mathcal{DB}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}'_{in}, \mathcal{U}'_{ext}, \mathcal{N}', \mathcal{L}')} \text{ CM-across-FireSingle}$$

$$\boxed{compressionAcrossCM(@\iota, \Sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')}$$

$$\frac{}{compressionAcrossCM(@\iota, [], \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = ([], [], \mathcal{N}, \mathcal{L})} \text{ CM-across-Compress-Empty}$$

$$\frac{singleCompressionAcrossCM(@\iota, \sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}') \\ compressionAcrossCM(@\iota, \Sigma, \Delta r, u_{cm}, \mathcal{N}', \mathcal{L}') = (\mathcal{U}_{cm''_{in}}, \mathcal{U}_{cm''_{ext}}, \mathcal{N}'', \mathcal{L}'')}{compressionAcrossCM(@\iota, \sigma :: \Sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}} \circ \mathcal{U}_{cm''_{in}}, \mathcal{U}_{cm'_{in}} \circ \mathcal{U}_{cm''_{ext}}, \mathcal{N}'', \mathcal{L}'')} \text{ CM-across-Compress}$$

$$\boxed{singleCompressionAcrossCM(@\iota, \sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')}$$

$$\frac{\begin{array}{c}\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \\ u_{cm} = \langle q(@\iota_q, \vec{t}_q), \mathit{Create}, \mathtt{eID}, \lambda_q\rangle \qquad q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi} \\ \forall i \in [1,n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma) \qquad \mathit{ruleargs}_p = \mathit{rID} :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \\ \mathtt{HrID}_p = \mathsf{hash}(\mathit{ruleargs}_p) \qquad \text{if } (\Gamma(q)[\mathit{type}] = \mathsf{event}) \text{ then } \flat_p = \mathsf{hash}(\lambda_q{:}3) \text{ else } \flat_p = \mathsf{hash}(\lambda_q) \\ \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p) \qquad u_{cm}{}' = \langle p(@\ell_p, \vec{x}_p)\sigma, \mathit{Create}, \mathtt{eID}, \lambda_p\rangle \\ ncm_p = (\langle @\iota_q, \mathtt{HrID}_p\rangle, \mathit{ruleargs}_p) \qquad \mathcal{N}' = \mathcal{N} \cup ncm_p \qquad lcm_p = (\lambda_p, \lambda_q) \\ \mathcal{L}' = \mathcal{L} \cup lcm_p \qquad \text{if } \sigma(@\ell_p) = @\iota_q \text{ then } \mathcal{U}_{cm'_{in}} = [u_{cm}'], \mathcal{U}_{cm'_{ext}} = [] \text{ else } \mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm}'] \end{array}}{\mathit{singleCompressionAcrossCM}(@\iota_q, \sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}', \mathcal{L}')}\ \text{CM-across-Create}$$

$$\frac{\begin{array}{c}\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots \\ u_{cm} = \langle q(@\iota_q, \vec{t}_q), \mathit{NCreate}, \mathtt{eID}, \lambda_q\rangle \\ q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi} \\ \forall i \in [1,n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma) \qquad \mathit{ruleargs}_p = \mathit{rID} :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \\ \mathtt{HrID}_p = \mathsf{hash}(\mathit{ruleargs}_p) \qquad \text{if } (\Gamma(q)[\mathit{type}] = \mathsf{event}) \text{ then } \flat_p = \mathsf{hash}(\lambda_q{:}3) \text{ else } \flat_p = \mathsf{hash}(\lambda_q) \\ \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p) \qquad u_{cm}{}' = \langle p(@\ell_p, \vec{x}_p)\sigma, \mathit{NCreate}, \mathtt{eID}, \lambda_p\rangle \\ \text{if } \sigma(@\ell_p) = @\iota_q \text{ then } \mathcal{U}_{cm'_{in}} = [u_{cm}'], \mathcal{U}_{cm'_{ext}} = [] \text{ else } \mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm}'] \end{array}}{\mathit{singleCompressionAcrossCM}(@\iota_q, \sigma, \Delta r, u_{cm}, \mathcal{N}, \mathcal{L}) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{N}, \mathcal{L})}\ \text{CM-across-NCreate}$$

F. EXAMPLE DEPENDENCY GRAPH

Figure 27 shows an example attribute-level dependency graph for the packet forwarding program in Figure 1. Based on Section 5.2, the equivalence keys are (packet:0, packet:2).

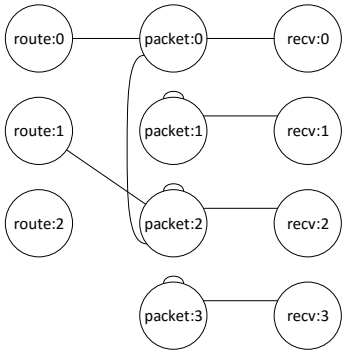

Figure 27: The attribute-level dependency graph for the packet forwarding program in Figure 1.

G. CORRECTNESS OF COMPRESSION

In order to prove that our online compression algorithm is correct – that it stores all the expected provenances and nothing more, we show that there is a bisimulation relation between network states for semi-naïve evaluation and online compression execution that shares storage across equivalence classes.

This section is organized as follows. First in Appendix G.1, we define a bisimulation relation between the network state for semi-naïve evaluation and online compression execution that shares storage *within* equivalence classes, and show that it holds after every pair of corresponding transition rules in both systems is fired. Next in Appendix G.2, we define a second bisimulation relation between the network state of online compression execution that shares storage *within* equivalence classes and online compression execution that shares storage *across* equivalence classes, and again show that it holds after every pair of corresponding transition rules in both systems is fired. In this way, we see that all provenances derived and stored by the semi-naïve evaluation is also derived and stored by the online compression execution that shares storage *across* equivalence classes and vice versa.

G.1 Bisimulation between semi-naïve evaluation and online compression execution

First, we show that there is a bisimulation between semi-naïve evaluation and online compression execution that shares storage *within* equivalence classes. In Appendix G.1.1, we formally define a relation $\boldsymbol{\mathcal{R}_C}$ between the network configuration $\mathcal{C}_{sn}$ of semi-naïve evaluation and the network configuration $\mathcal{C}_{cm}$ of online compression execution and show that the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ defines a bisimulation between the two executions. Next, in Appendix G.1.2, we show that every time the semi-naïve evaluation takes a step,

G.1.1 Relating network states

We define relations between constructs for semi-naïve evaluation and online compression execution that shares storage *within* equivalence classes.

Relating a single update ($\Gamma \vdash u_{sn} \sim_u u_{cm}$).

In the base case, when a tuple ev arriving on a node is an event relation, rule U-Base states that the update for semi-naïve evaluation is simply the event tuple itself, while the update for online compression evaluation is the event tuple, the flag that tells us whether to maintain provenances, and the hash of ev.

In the inductive case when tuple Q arriving on a node is not an event relation, then tuple Q must have been been derived from a previous rule that has already been fired. Tuple Q triggers another rule on the node to derive a new tuple P. Rule U-Ind states that the update for semi-naïve evaluation is the entire provenance tree for P which has the provenance tree for Q as a subtree, while the update for online compression evaluation is the tuple P, the flag that tells us whether to maintain provenances (which must be the same as that of the update for Q) the hash of the event tuple the triggered program execution, and the unique identifier for the provenance of the rule that derived P.

Relating multiple updates ($\Gamma \vdash \mathcal{U}_{sn}\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ \mathcal{U}_{cm}$).

The base case is when both $\mathcal{U}_{sn}$ and $\mathcal{U}_{cm}$ are empty sets.

In the inductive case, every update u_{sn} in $\mathcal{U}_{sn}$ must be related to an update u_{cm} in $\mathcal{U}_{cm}$ according to the relation $u_{sn}\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ u_{cm}$ and vice versa.

Relating a provenance tree to an ordered list of rule provenances ($\Gamma \vdash tr \sim_d yl$).

Rule $\sim_d$-Base states that when an incoming event tuple triggers execution of the first rule in the program to derive tuple P, each construct in the provenance tree for P can be related to a construct in the rule provenance.

Rule $\sim_d$-Ind states that if the incoming tuple Q is not an event tuple and its provenance tree relates to an order list of rule provenances and triggers execution of a rule rID in the program to derive tuple P, then the provenance tree for P can be related to the list of rule provenances for tuple Q with the rule provenance for the rID appended to the end of the list.

Relating provenance trees to rule provenances ($\Gamma \vdash \mathcal{M} \approx_d \Upsilon$).

The base case is when both $\mathcal{M}$ and Υ are empty sets.

In the inductive case, every provenance tree tr in $\mathcal{M}$ relates to an ordered list of rule provenances yl, and every element of yl can be found in Υ.

Determining an potential update ($\boldsymbol{DQ}, \Gamma \vdash u_{cm} \Rightarrow ruleExec, u_{cm}'$).

Given an update u_{cm} for tuple Q and the program execution, rule $\Rightarrow$-Update determines a potential update u_{cm}' that can be generate given program DQ and the tuple associated with u_{cm}, as well as the corresponding rule provenance.

Determining future provenances generated from a single update ($\Gamma \vdash u_{cm} \looparrowright yl$).

Given an update u_{cm} for tuple Q and the program execution, rule $\looparrowright$-Ind can be repeatedly applied to determine the allowable future updates according to the program, and an ordered list of rule provenances for the allowable future updates.

Determining future provenances generated by multiple updates ($\Gamma \vdash \mathcal{U}_{cm} \curvearrowright \Upsilon$).

Given a set of updates $\mathcal{U}_{cm}$ and the program, repeated application of rule $\curvearrowright$-Update derives all the rule provenances that could possibly be generated by the updates in $\mathcal{U}_{cm}$.

Relating a provenance tree to a tuple provenance element ($\Gamma, \Upsilon \vdash \mathsf{interest}(tr) \sim_{prov} prov$).

Given that tuple P is an instance of a relation of interest, and given that the provenance tree of P relates via $\sim_d$ to an ordered list of tuples, a tuple provenance node for P stores the location specifier of P, the hash of the primary keys of

P, the hash of the primary keys of the event tuple that triggered the execution sequence that derived P, and the lookup key for the tail element of the ordered list.

Relating provenance trees to current and future rule provenances ($\Gamma, \boldsymbol{DQ}, \mathcal{U}_{cm} \vdash \mathcal{M}\ \boldsymbol{\mathcal{R}_{re}}\ \Upsilon$).

Given a set of provenance trees $\mathcal{M}$ that are generated by semi-naïve evaluation, a set of rule provenances Υ have already been generated by online compression evaluation, and that online compression evaluation will eventually use the updates in $\mathcal{U}_{cm}$ to generate a set of rule provenances Υ^F, rule Relate-Rule-Prov derives that all provenance trees in $\mathcal{M}$ relates to the existing rule provenances Υ given the updates $\mathcal{U}_{cm}$.

Relating a set of provenance trees to a set of tuple provenances ($\Gamma, \Upsilon \vdash \mathcal{M}_{prov}\ \boldsymbol{\mathcal{R}_{prov}}\ \Upsilon_{prov}$).

In the base case, both $\mathcal{M}_{prov}$ and Υ_{prov} are empty sets, thus $\approx_{prov}$ trivially relates the empty sets.

In the inductive case, every element in $\mathcal{M}_{prov}$ is a provenance tree $\mathsf{interest}(tr)$ for an instance of a relation of interest, that relates via $\sim_{prov}$ to a tuple provenance node in Υ_{prov} and vice versa.

Relating the configurations for semi-naïve to online compression evaluation ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$).

Most of the constructs used to define the network configurations for semi-naïve evaluation and online compression evaluation that shares storage *within* equivalence classes are identical, except for the way updates are handled and how provenance is maintained.

Rule Relate-Config relates the updates from both evaluations using the relation $\boldsymbol{\mathcal{R}_{\mathcal{U}}}$.

Rule Relate-Rule-Prov relates $\mathcal{M}$, the set of provenance trees of all tuples derived by the semi-naïve evaluation to Υ, the set of rule provenances generated by the online compression evaluation. Because updates may be processed out of order, this rule makes use of the updates that have yet to be fired by online compression evaluation to show that the sets of provenances in both evaluations will eventually correspond.

The relation $\boldsymbol{\mathcal{R}_{prov}}$ makes use of the set of future rule provenances that will eventually be generated by the updates to relate the provenance trees of all tuples of relations of interest derived by the semi-naïve evaluation to the tuple provenances of relations of interest derived by the online compression evaluation that shares storage *within* equivalence classes.

Because the every provenance tree derived and stored by the semi-naïve evaluation will eventually corrspond to some rule provenance(s) derived and stored by the online compression evaluation and vice versa, the two evaluations always store the exact same provenances when execution terminates.

$\boxed{\Gamma \vdash u_{sn} \sim_u u_{cm}}$

$$\frac{\Gamma[e][type] = \mathsf{event} \qquad heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma) \qquad \mathtt{eID} = \textsc{TupleHash}(e(@\iota_e, \vec{t_e}), \Gamma)}{\Gamma \vdash e(@\iota_e, \vec{t_e})) \sim_u \langle e(@\iota_e, \vec{t_e})), \mathsf{createFlag}, \mathtt{eID}, \mathsf{id}(\emptyset, \emptyset, heq)\rangle}\ \textsc{u-Base}$$

$$\frac{\begin{array}{c}\Gamma \vdash tr_q : q(@\iota_q, \vec{t_q}) \sim_u \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle \qquad \forall i \in [1,n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_q, \vec{t}_{bi}), \Gamma) \\ ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \qquad \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \mathsf{hash}(\lambda_q))\end{array}}{\Gamma \vdash (rID, p(@\iota_p, \vec{t_p}), tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\iota_q, \vec{t}_{b1}) :: \cdots :: b_n(@\iota_q, \vec{t}_{bn})) : p(@\iota_p, \vec{t_p}) \sim_u \langle p(@\iota_p, \vec{t_p})), \mathsf{createFlag}, \mathtt{eID}, \lambda_p\rangle}\ \textsc{u-Ind}$$

$\boxed{\Gamma \vdash \mathcal{U}_{sn} \sim_u \mathcal{U}_{cm}}$

$$\frac{}{\Gamma \vdash []\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ []}\ \mathcal{U}\textsc{-Base} \qquad \frac{\Gamma \vdash u_{sn} \sim_u u_{cm} \qquad \Gamma \vdash \mathcal{U}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{U}_{cm}}{\Gamma \vdash u_{sn} :: \mathcal{U}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ u_{cm} :: \mathcal{U}_{cm}}\ \mathcal{U}\textsc{-Ind}$$

$\boxed{\Gamma \vdash tr \sim_d yl}$

$$\frac{\begin{array}{c}tr_p = (rID, p(@\iota_p, \vec{t_p}), e(@\iota_e, \vec{t_e}), b_1(@\iota_e, \vec{t}_{b1}) :: \cdots :: b_n(@\iota_e, \vec{t}_{bn})) \\ heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma) \qquad \lambda_e = \mathsf{id}(\emptyset, \emptyset, heq) \qquad \forall i \in [1,n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_e, \vec{t}_{bi}), \Gamma) \\ ruleargs_p = rID :: \iota_e :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \qquad \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \lambda_p = \mathsf{id}(@\iota_e, \mathtt{HrID}_p, heq)\end{array}}{\Gamma \vdash tr_p \sim_d \langle \lambda_p, ruleargs_p, \lambda_e\rangle}\ \sim_d\textsc{-Base}$$

$$\frac{\begin{array}{c}\Gamma \vdash tr_q{:}q(@\iota_q, \vec{t_q})) \sim_d yl_\rho :: \langle \lambda_q, ruleargs_q, \lambda_\rho\rangle \qquad \forall i \in [1,n], \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_q, \vec{t}_{bi}), \Gamma) \\ ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n \qquad \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \mathsf{hash}(\lambda_q))\end{array}}{\begin{array}{c}\Gamma \vdash (rID, p(@\iota_p, \vec{t_p}), tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\iota_q, \vec{t}_{b1}) :: \cdots :: b_n(@\iota_q, \vec{t}_{bn})) \\ \sim_d yl_\rho :: \langle \lambda_q, ruleargs_q, \lambda_\rho\rangle :: \langle \lambda_p, ruleargs_p, \lambda_q\rangle\end{array}}\ \sim_d\textsc{-Ind}$$

$\boxed{\Gamma \vdash \mathcal{M} \approx_d \Upsilon}$

$$\frac{}{\Gamma \vdash \{\} \approx_d \{\}}\ [d]\textsc{-Base} \qquad \frac{\Gamma \vdash tr \sim_d yl \qquad \Gamma \vdash \mathcal{M} \sim_d \Upsilon}{\Gamma \vdash \mathcal{M} \cup tr \approx_d \Upsilon \cup yl}\ [d]\textsc{-Ind}$$

$\boxed{DQ, \Gamma \vdash u_{cm} \Rightarrow ruleExec, u_{cm}{}'}$

$$\frac{u_{cm_q} = \langle q(@\iota_q, \vec{t}_q),\, Create, \texttt{eID}, \lambda_q\rangle \qquad rID\ p(@\ell_p, \vec{x}_p) \coloneq q(@\ell_q, \vec{x}_q), b_1(@\iota_q, \vec{x}_{b1}), \cdots, b_n(@\iota_q, \vec{x}_{bn}), \cdots \in DQ}{\begin{array}{c} q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q) \qquad \mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi} \qquad \forall i \in [1,n], \texttt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma) \\ ruleargs_p = rID :: \sigma(\iota_q) :: \texttt{vID}_1 :: \cdots :: \texttt{vID}_n \qquad ruleExec_q = \langle \lambda_p, ruleargs_p, \lambda_q\rangle \\ \texttt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \flat_p = \lambda_q{:}3 \qquad \lambda_p = \mathsf{id}(@\iota_q, \texttt{HrID}_p, \flat_p) \qquad u_{cm_p} = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \texttt{eID}, \lambda_p\rangle \end{array}}{DQ, \Gamma \vdash u_{cm_q} \Rightarrow ruleExec_p, u_{cm_p}}\ \Rightarrow\text{-Update}$$

$\boxed{\Gamma \vdash u_{cm} \looparrowright yl}$

$$\frac{}{DQ, \Gamma \vdash u_{cm} \looparrowright []}\ \looparrowright\text{-Base} \qquad \frac{DQ, \Gamma \vdash u_{cm} \Rightarrow ruleExec, u_{cm}' \qquad DQ, \Gamma \vdash u_{cm}' \looparrowright yl'}{DQ, \Gamma \vdash u_{cm} \looparrowright [ruleExec] \circ yl'}\ \looparrowright\text{-Ind}$$

$\boxed{\Gamma \vdash \mathcal{U}_{cm} \curvearrowright \Upsilon}$

$$\frac{}{DQ, \Gamma \vdash \{\} \curvearrowright \{\}}\ \curvearrowright\text{-Base} \qquad \frac{\forall i \in [1,n], DQ, \Gamma \vdash u_{cm} \looparrowright yl_i \qquad DQ, \Gamma \vdash \mathcal{U}_{cm} \curvearrowright \Upsilon}{DQ, \Gamma \vdash \mathcal{U}_{cm} \cup \bigcup_{i=1}^{n} \mathcal{U}_{cm_i} \curvearrowright \Upsilon \cup \bigcup_{i=1}^{n} yl_i}\ \curvearrowright\text{-Ind}$$

$\boxed{\Gamma, DQ, \mathcal{U}_{cm} \vdash \mathcal{M}\ \boldsymbol{\mathcal{R}}_{\mathbf{re}}\ \Upsilon}$

$$\frac{\Gamma, DQ \vdash \mathcal{U}_{cm} \curvearrowright \Upsilon^F \qquad \Gamma \vdash \mathcal{M} \approx_d \Upsilon \cup \Upsilon^F}{\Gamma, DQ, \mathcal{U}_{cm} \vdash \mathcal{M}\ \boldsymbol{\mathcal{R}}_{\mathbf{re}}\ \Upsilon}\ \textsc{Relate-Rule-Prov}$$

$\boxed{\Gamma, \Upsilon \vdash \mathsf{interest}(tr) \sim_{prov} prov}$

$$\frac{\begin{array}{c} \textsc{EventOf}(tr_r{:}r(@\iota_r, \vec{t}_r)) = e(@\iota_e, \vec{t}_e) \\ \texttt{eID} = \textsc{TupleHash}(e(@\iota_e, \vec{t}_e), \Gamma) \qquad \Gamma \vdash tr_r{:}r(@\iota_r, \vec{t}_r) \sim_d yl_q :: \langle \lambda_r, ruleargs_r, \lambda_q\rangle \\ yl_q :: \langle \lambda_r, ruleargs_r, \lambda_q\rangle \subseteq \Upsilon \qquad \texttt{tID} = \textsc{TupleHash}(r(@\iota_r, \vec{t}_r), \Gamma) \end{array}}{\Gamma, \Upsilon \vdash \mathsf{interest}(tr_r{:}r(@\iota_r, \vec{t}_r)) \sim_{prov} \langle @\iota_r, \texttt{tID}, \texttt{eID}, \lambda_r\rangle}\ \textsc{Relate-Prov}$$

$\boxed{\Gamma, \Upsilon \vdash \mathcal{M}_{prov} \approx_{prov} \Upsilon_{prov}}$

$$\frac{}{\Gamma, \Upsilon \vdash \{\} \approx_{prov} \{\}}\ \approx_{prov}\text{-Base} \qquad \frac{\Gamma, \Upsilon \vdash \mathsf{interest}(tr_r{:}r(@\iota_r, \vec{t}_r)) \sim_{prov} prov \qquad \Gamma, \Upsilon \vdash \mathcal{M}_{prov} \approx_{prov} \Upsilon_{prov}}{\Gamma, \Upsilon \vdash \mathcal{M}_{prov} \cup \mathsf{interest}(tr_r{:}r(@\iota_r, \vec{t}_r)) \approx_{prov} \Upsilon_{prov} \cup prov}\ \approx_{prov}\text{-Ind}$$

$\boxed{\Gamma, DQ, \mathcal{U}_{cm}, \Upsilon \vdash \mathcal{M}_{prov}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\ \Upsilon_{prov}}$

$$\frac{}{\Gamma, DQ, \mathcal{U}_{cm}, \Upsilon \vdash \{\}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\ \{\}}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\text{-Base} \qquad \frac{\Gamma, DQ \vdash \mathcal{U}_{cm} \looparrowright \Upsilon^F \qquad \Gamma, \Upsilon \cup \Upsilon^F \vdash \Upsilon_{prov} \approx_{prov} \mathcal{M}_{prov}}{\Gamma, DQ, \mathcal{U}_{cm}, \Upsilon \vdash \mathcal{M}_{prov}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\ \Upsilon_{prov}}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\text{-Ind}$$

$\boxed{\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}\ \mathcal{C}_{cm}}$

$$\frac{\begin{array}{c} \forall i \in [1,N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i}\rangle \\ \forall i \in [1,N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i}\rangle \\ \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{U}}}\ \mathcal{Q}_{cm} \qquad \forall i \in [1,N], \Gamma \vdash \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{U}}}\ \mathcal{U}_{cm_i} \qquad \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i} \\ \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}}_{\mathbf{re}}\ \bigcup_{i=1}^{N} \Upsilon_i \qquad \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}}_{\mathbf{prov}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i} \end{array}}{\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}}\ \textsc{Relate-Config}$$

G.1.2 Semi-naïve evaluation simulates online compression execution

We show that semi-naïve evaluation simulates online compression execution that shares storage *within* equivalence classes. To do so, for each transition rule for semi-naïve evaluation, we state and prove a lemma that shows that the rule has a corresponding counterpart in online compression execution. If initially the network configuration for both systems relate, after

semi-naïve evaluation steps to a new configuration, then online compression execution is also able to step to a corresponding new configuration. We present the lemmas and their proofs below.

Multi-step transition: semi-naïve simulates online compression (Lemma 6).

We define $\mathcal{C}_{init}$ to be the initial network configuration when no updates have been fired and not provenance has been stored. We show that given any Semi-Naïve evaluation that transitions from $\mathcal{C}_{init}$ to $\mathcal{C}_{sn_{k+1}}$ in k steps, there exists an Online Compression evaluation that also transitions from $\mathcal{C}_{init}$ to $\mathcal{C}_{cm_{k+1}}$ in k steps, and furthermore that the network configurations relate (i.e. $\mathcal{C}_{sn_{k+1}}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm_{k+1}}$).

To prove this lemma, we use induction over k. In the base case when $k = 0$, $\mathcal{C}_{sn_1} = \mathcal{C}_{init} = \mathcal{C}_{cm_1}$, so it is obvious that $\mathcal{C}_{sn_1}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm_1}$. In the inductive case when $k = m + 1$, $\mathcal{C}_{init}$ transitions to $\mathcal{C}_{sn_k}$ in m steps, thus by the induction hypothesis that $\mathcal{C}_{sn_k}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm_k}$. Now using Single-step transition: semi-naïve simulates online compression (Lemma 7), we see that given $\mathcal{C}_{sn_k} \rightarrow^{k}_{SN} \mathcal{C}_{sn_{k+1}}$, there exists $\mathcal{C}_{cm_{k+1}}$ s.t. $\mathcal{C}_{sn_{k+1}}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm_{k+1}}$.

Single-step transition: semi-naïve simulates online compression (Lemma 7).

Given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$), if the semi-naïve evaluation takes a step and transitions to $\mathcal{C}_{sn}{}'$, then when online compression execution takes a step to $\mathcal{C}_{cm}{}'$, these new network configurations again relate ($\mathcal{C}_{sn}{}'\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}{}'$).

The proof uses the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$ and inversion over the transition rules for the network configuration ($\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}{}'$).

Case A: the last rule that derived $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}{}'$ was SN-NodeStep**.**

The overall network configurations in both systems took a step because some state $\mathcal{S}_{sn_i}$ in $\mathcal{C}_{sn}$ transitioned to a new state $\mathcal{S}_{sn_i}'$ with additional external updates $\mathcal{U}_{sn_i}'$. We use Single-step transition per node: semi-naïve simulates online compression 8 and the corresponding Online Compression rule CM-NodeStep to obtain the goal.

Case B: the last rule that derived $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}{}'$ was SM-DeQueue**.**

The overall network configurations in both systems took a step because external updates in $\mathcal{C}_{sn}$ were sent to different nodes in the network based on their location specifier. Since external updates in $\mathcal{C}_{sn}$ correspond to those in $\mathcal{C}_{cm}$, by CM-DeQueue we have our goal.

Single-step transition per node: semi-naïve simulates online compression (Lemma 8).

Given two related configurations ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$), if state $\mathcal{S}_{sn\ell}$ in $\mathcal{C}_{sn}$ transitioned to $\mathcal{S}_{sn\ell}'$ with external updates $\mathcal{U}_{sn\ell}'$, then the corresponding state $\mathcal{S}_{cm\ell}$ in $\mathcal{C}_{cm}$ transitioned to $\mathcal{S}_{cm\ell}'$ with external updates $\mathcal{U}_{cm\ell}'$.

The proof uses the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$ and inversion over the transition rules for the individual nodes ($\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn\ell}', \mathcal{U}_{sn_{ext}}'$).

Case A: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn\ell}', \mathcal{U}_{sn_{ext}}'$ was SN-Event**.**

Rule SN-Event popped off an event in $\mathcal{E}$ and fired an update. It is easy to relate the respective updates for both systems show that the resultant list of internal events and updates correspond. The provenance trees in $\mathcal{S}_{sn\ell}$ and $\mathcal{S}_{sn\ell}'$ are the same.

Case B: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn\ell}', \mathcal{U}_{sn_{ext}}'$ was SN-RuleFire-Fast**.**

Rule SN-RuleFire-Fast takes in an update and substitutions for a rule, then generates a new update based on these arguments. Thus, the set of provenances and updates for fast-changing tuples is incremented. By *fireRulesSN* simulates *fireRulesCM* (Lemma 10) and CM-RuleFire-Fast, we obtain the desired conclusion.

Case C: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn\ell}', \mathcal{U}_{sn_{ext}}'$ was SN-RuleFire-Interest**.**

Rule SN-RuleFire-Interest takes as argument an update that contains a provenance tree tr_r:*res*, in which *res* is a tuple that is an instance of a relation of interest as an argument. It saves tr_r:*res* in the set of tuple provenances. No new updates nor new rule provenance are generated.

Since provenance tree tr_r:*res* is an update, thus the semi-naïve evaluation has already stored tr_r:*res* in the set of derived provenance trees $\mathcal{M}$ and the set of provenances for relations of interests $\mathcal{M}_{prov}$. By relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$, therefore rule provenances that correspond to tr_r:*res* are either already stored in Υ, or will eventually be generated. Now we apply rule Relate-Prov to show that we can store tr_r:*res* in Υ_{prov}. Since only the set of tuple provenances ($\mathcal{M}_{prov}$ and Υ_{prov}) is updated by rule SN-RuleFire-Interest, thus the updated network states for both executions again relate.

fireRulesSN **simulates** *fireRulesCM* **(Lemma 10).**

Given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$), *fireRulesSN*$(@_{\iota\ell}, \Delta\bar{DQ}, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell)$ takes in an update $u_{sn\ell}$, a subset of the program $\bar{DQ}$ and returns new updates and provenance trees.

This lemma is proved using induction over $|\bar{DQ}|$. In the base case then there are no rules to be fired, $\mathcal{C}_{sn}{}' = \mathcal{C}_{sn}$ and $\mathcal{C}_{cm}{}' = \mathcal{C}_{cm}$, so the conclusion is trivially true. In the inductive case when $|\bar{DQ}| = k+1$, the last rule fired was SN-Seq. By inversion on that rule we see that we should use *fireSingleRuleSN* simulates *fireSingleRuleCM* (Lemma 11), the induction hypothesis, and then CM-Seq to obtain the goal.

fireSingleRuleSN **simulates** *fireSingleRuleCM* **(Lemma 11).**

Given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$), *fireSingleRuleSN*$(@_{\iota\ell}, \Delta r, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell)$ takes in an update $u_{sn\ell}$, a rule in the program DQ, and returns new updates and provenance trees.

We prove the lemma using Lemma *derivationSN* simulates *compressionCM* (Lemma 12) and CM-FireSingle.

derivationSN **simulates** *compressionCM* **(Lemma 12).**

Given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$), $derivationSN(@_{\iota_\ell}, \Sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell)$ takes in an update $u_{sn\ell}$, a rule r in the program DQ, a subset Σ of all possible substitutions for r and returns new updates and provenance trees.

This lemma is proved using induction over $|\Sigma|$. In the base case then there are no possible substitutions and rule r cannot be fired, thus $\mathcal{C}_{sn}{}' = \mathcal{C}_{sn}$ and $\mathcal{C}_{cm}{}' = \mathcal{C}_{cm}$, and the conclusion is trivially true. In the inductive case when $|\Sigma| = k + 1$, the last rule fired was SN-Subst. By inversion on that rule we see that we should use *singleDerivSN* simulates *singleCompressionCM* (Lemma 13) the induction hypothesis, and then CM-Subst to obtain the goal.

singleDerivSN **simulates** *singleCompressionCM* **(Lemma 13).**

This is the key lemma that deals with updating the set of rule provenances. The proof is fairly complicated due to potential out of order executions. Because semi-naïve evaluation stores one provenance tree per execution while online compression execution only stores one set of rule provenances per equivalence class, out of order executions may result in the provenances in the systems not having an obvious correspondence during program execution. Consequently, our proof need to argue that the missing rule provenances will eventually be generated by the online compression execution.

The lemma shows that given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$), $singleDerivSN(@_{\iota_\ell}$, σ, Δr, $u_{sn\ell}$, $\mathcal{M}_\ell$) takes in an update $u_{sn\ell}$, a rule r in the program DQ, a substitution σ for r, and returns a new update $u_{sn\ell}'$ and a new provenance tree.

There are several cases to consider:

Case I: $u_{sn\ell}$ **represents a tuple that is an instance of the input event relation.**

By the rules Semi-Naïve evaluation, the last transition rule executed was SN-SingleSubst-Event. Therefore by inversion on the rule, exists an input event tuple ev s.t. $u_{sn\ell} = ev$ and exists a provenance tree $tr_p{:}P$ s.t. $u_{sn\ell}' = tr_p{:}P$ and ev is a subformula of $tr_p{:}P$.

Case A: $u_{cm\ell}.\mathsf{createFlag} = Create$**.**

Using the constructs obtained from inversion, we fire the corresponding cnline compression rule CM-Create to return an update $u_{cm\ell}'$ and a new rule provenance $ruleExec_p$. Because only one rule in DQ has been fired so far, it is easy to see that provenances $tr_p{:}P$ and $ruleExec_p$ relate and furthermore that updates $u_{sn\ell}'$ and $u_{cm\ell}'$ relate. We show that the new provenances added to both systems relate. Since value of createFlag is *Create*, $ruleExec_p$ was created and stored in $\mathcal{C}_{cm}$. We use the above facts and $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ to show that the network configurations of both executions after firing SN-SingleSubst-Event and CM-Create again relate.

We use $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ and the above facts about the new update and rule provenance generated to show that the network configurations of both executions after firing SN-SingleSubst-Event and CM-Create will again relate.

Case B: $u_{cm\ell}.\mathsf{createFlag} = NCreate$**.**

By the constructs obtained from inversion, we fire the corresponding online compression rule CM-Create to return an update $u_{cm\ell}'$ and a new rule provenance $ruleExec_p$. Because only one rule in DQ has been fired so far, it is easy to see that provenances $tr_p{:}P$ and $ruleExec_p$ relate and furthermore that updates $u_{sn\ell}'$ and $u_{cm\ell}'$ relate. We show that the new provenances added to both systems relate. Since value of createFlag is *NCreate*, there are two cases to consider. (1) $ruleExec_p$ is already stored in $\mathcal{C}_{cm}$. By examining the rules for online compression execution, in the past some update $u_{cm\ell}^{\nu}$ (where $u_{cm\ell}^{\nu} = u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$) had already been fired, causing $ruleExec_p$ to be created and stored in $\mathcal{C}_{cm}$. Because the network configurations of both systems relate, thus $tr_p{:}P$ is already stored in $\mathcal{C}_{sn}$ as well. Therefore previous updates already generate provenances $tr_p{:}P$ and $ruleExec_p$ and thus when rules SN-SingleSubst-Event and CM-Create were fired no new provenances were stored. (2) $ruleExec_p$ is not stored in $\mathcal{C}_{cm}$. By examining the rules for online compression execution, there is an update $u_{cm\ell}^{\nu}$ (where $u_{cm\ell}^{\nu} = u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$) that has not been fired yet and is still stored in the set of updates in $\mathcal{C}_{cm}$. However the set of rule provenances in $\mathcal{C}_{sn}$ is updated to include $tr_p{:}P$. We use $u_{cm\ell}^{\nu}$ to argue that in the future $ruleExec_p$ will be created and stored, thus the rule provenances in both systems still relate.

We use $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ and the above facts about the new update and rule provenance generated to show that the network configurations of both executions after firing SN-SingleSubst-Event and CM-Create will again relate.

Case II: $u_{sn\ell}$ **represents a tuple that is an instance of a fast-changing relation/ relation of interest.**

By the rules semi-naïve evaluation, the last transition rule executed was SN-SingleSubst-Fast. Therefore by inversion on that rule, exists a provenance tree $tr_q{:}Q$ s.t. $u_{sn\ell} = tr_q{:}Q$ and exists a provenance tree $tr_p{:}P$ s.t. $u_{sn\ell}' = tr_p{:}P$ and $tr_q{:}Q$ is a subtree in $tr_p{:}P$.

Case A: $u_{cm\ell}.\mathsf{createFlag} = Create$**.**

By the transition rules semi-naïve evaluation, $tr_q{:}Q$ is stored in $\mathcal{C}_{sn}$. Thus given the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ there exists a list of rule provenances yl_q that relates to $tr_q{:}Q$. Since createFlag = *Create*, rule provenances are created during this online compression execution, so yl_q is concretely stored in the set of rule provenances in $\mathcal{C}_{cm}$.

Using the constructs obtained by inversion on SN-SingleSubst-Fast, we fire the corresponding rule CM-Create and obtain the new rule provenance $ruleExec_p$ and new update $u_{cm\ell}'$. $ruleExec_p$ stores the provenance for the execution of rule r triggered by tuple Q that uses substitution σ. Given that $u_{sn\ell}$ and $u_{cm\ell}$ relate, it is easy to see that $u_{sn\ell}'$ also relates to $u_{cm\ell}'$.

We show that the new provenances added to both systems relate. Since createFlag is *Create*, rule provenances are created during this online compression execution, so $ruleExec_p$ is concretely stored in $\mathcal{S}_{cm\ell}$. Using the above results we show that $tr_p{:}P$ and $yl_q :: ruleExec_p$ relate and $yl_q :: ruleExec_p$ is concretely stored in $\mathcal{C}_{cm}$. We use the above facts and $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ to show that the network configurations of both executions after firing

SN-SingleSubst-Fast and CM-Create again relate.

Case B: $u_{cm\ell}$.createFlag = *NCreate*.

By the transition rules Semi-Naïve evaluation, $tr_q{:}Q$ is stored in $\mathcal{C}_{sn}$. Thus given the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ there exists a list of rule provenances yl_q that relates to $tr_q{:}Q$.

Using the constructs obtained by inversion on SN-SingleSubst-Fast, we fire the corresponding rule CM-NCreate and obtain the new rule provenance $ruleExec_p$ and new update $u_{cm'_\ell}$. $ruleExec_p$ stores the provenance for the execution of rule r triggered by tuple Q that uses substitution σ. Given that $u_{sn\ell} and u_{cm\ell}$ relate, it is easy to see that $u_{sn'_\ell}$ also relates to $u_{cm'_\ell}$.

We show that the new provenances added to both systems relate. Since value of createFlag is *NCreate*, there are two cases to consider. (1) yl_q is already stored entirely within $\mathcal{C}_{cm}$. If $ruleExec_p$ is also stored in $\mathcal{C}_{cm}$, then the rule provenances in both system configurations again relate. If $ruleExec_p$ is not stored in $\mathcal{C}_{cm}$, By examining the rules for online compression execution, there is an update $u_{cm^\nu_\ell}$ (where $u_{cm^\nu_\ell} = u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$) that has not been fired yet and is still stored in the set of updates in $\mathcal{C}_{cm}$. However the set of rule provenances in $\mathcal{C}_{sn}$ is updated to include $tr_p{:}P$. We use $u_{cm^\nu_\ell}$ to argue that in the future $ruleExec_p$ will be created and stored, thus the rule provenances in both systems still relate. (2) yl_q is not stored entirely within $\mathcal{C}_{cm}$. By $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ part of yl_q is contained in $\mathcal{C}_{cm}$ (call it yl_A) and there is some update u_{cm^ν} (where u_{cm^ν}.createFlag = *Create*) that generates yl_B, where $yl_q = yl_A \circ yl_B$. Since u_{cm^ν} will eventually cause updates $u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$, $u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$ and rule provenance$ruleExec_p$ to be generated as well, therefore the missing rule provenances $yl_B :: ruleExec_p$ will eventually be created and stored. Thus the rule provenances in both systems still relate after the transition rules have been fired.

Lemma 6 (Multi-step transition: semi-naïve simulates online compression)**.**
$\forall k \in \mathbb{N}$,
$\mathcal{C}_{init} \rightarrow^k_{SN} \mathcal{C}_{sn_{k+1}}$
implies
$\exists \mathcal{C}_{cm_{k+1}}$ *s.t.*
$\mathcal{C}_{init} \nearrow^k_{CM} \mathcal{C}_{cm_{k+1}}$
and $\mathcal{C}_{cm_{k+1}}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{sn_{k+1}}$.

Proof. By induction over k.

Base Case: $k = 0$**.**
By assumption,
(b1) $\mathcal{C}_{init} \rightarrow^0_{SN} \mathcal{C}_{init}$
We define:
(b2) the network configuration for online compression evaluation to be $\mathcal{C}_{init}$
Thus we have
(b3) $\mathcal{C}_{init} \nearrow^0_{CM} \mathcal{C}_{init}$
By Rule Relate-Config and since no provenances are stored in either configuration,
(b4) $\mathcal{C}_{init}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{init}$
By (b2) and (b4),
The conclusion follows

Inductive Case: $k = m + 1$**.**
Given $\mathcal{C}_{init} \rightarrow^m_{SN} \mathcal{C}_{sn_k}$, by I.H. we have
(i1) $\exists \mathcal{C}_{cm_m}$ s.t.
$\mathcal{C}_{init} \nearrow^m_{CM} \mathcal{C}_{cm_{m+1}}$
and $\mathcal{C}_{cm_{m+1}}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{sn_{m+1}}$.
By assumption we have
(i2) $\mathcal{C}_{sn_k} \rightarrow_{SN} \mathcal{C}_{sn_{k+1}}$
Using (i1) and $\mathcal{C}_{sn_k} \rightarrow_{SN} \mathcal{C}_{sn_{k+1}}$ we apply
Single-step transition: semi-naïve simulates online compression (Lemma 7) to obtain:
(i3) $\exists \mathcal{C}_{cm_{k+1}}$ s.t.
$\mathcal{C}_{init} \nearrow^k_{CM} \mathcal{C}_{cm_{k+1}}$
and $\mathcal{C}_{cm_{k+1}}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{sn_{k+1}}$
By (i2) and (i3),
The conclusion follows

□

Lemma 7 (Single-step transition: semi-naïve simulates online compression)**.**
$\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$
and $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn'}$
implies
$\exists \mathcal{C}_{cm'}$ *s.t.*

$\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$
and $\mathcal{C}_{sn}' \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{C}_{cm}'$.

Proof.
Assume that
(1) $\mathcal{C}_{sn} \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{C}_{cm}$
(2) $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$
By inversion on the rules (2),
$\mathcal{C}_{sn} = \mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$
$\mathcal{C}_{cm} = \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
$\forall i \in [1, N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1, N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn} \ \boldsymbol{\mathcal{R}_{\mathcal{U}}} \ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N], \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i} \ \boldsymbol{\mathcal{R}_{\mathcal{U}}} \ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i \ \boldsymbol{\mathcal{R}_{\mathbf{re}}} \ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i} \ \boldsymbol{\mathcal{R}_{\mathbf{prov}}} \ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$
By inversion over the rules for $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$, we have the following cases:

Case A: the last rule that derived $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$ was SN-NodeStep**.**
By inversion we have
(a1) $\mathcal{S}_{sn_i} \hookrightarrow \mathcal{S}_{sn_i}', \mathcal{U}_{sn_i}'$
(a2) $\forall j \in [1, n] \wedge j \neq i$, $\mathcal{S}_{sn_j}' = \mathcal{S}_{sn_j}$
By (1) and (a1) we apply
Single-step transition per node: semi-naïve simulates online compression (Lemma 8) to obtain
(a3) $\exists \mathcal{U}_{cm_i}'$, $\exists \mathcal{S}_{cm_\ell}'$ s.t.
$\mathcal{S}_{cm_i} \hookrightarrow \mathcal{S}_{cm_i}', \mathcal{U}_{cm_i}'$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_i}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_i}' \cdots \mathcal{S}_{sn_N} \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm_i}' \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_i}' \cdots \mathcal{S}_{cm_N}$.
Define
(a4) $\mathcal{C}_{cm}' \triangleq \mathcal{Q}_{cm} \circ \mathcal{U}_{cm_i}' \rhd \mathcal{S}_{cm_1}' \cdots \mathcal{S}_{cm_i}' \cdots \mathcal{S}_{cm_N}'$
where $\forall j \in [1, N] \wedge j \neq \ell$, $\mathcal{S}_{cm_j}' = \mathcal{S}_{cm_j}$.
Apply CM-NodeStep to obtain
(a5) $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$
By (a3) and (a5),
The conclusion holds.

Case B: the last rule that derived $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$ was SN-DeQueue**.**
By inversion we have
(b1) $\mathcal{C}_{sn}' = \mathcal{Q}_{sn}' \rhd (\mathcal{S}_{sn_1} \circ \mathcal{Q}_{sn_1}) \cdots (\mathcal{S}_{sn_N} \circ \mathcal{Q}_{sn_N})$
(b2) $\mathcal{Q}_{sn} = \mathcal{Q}_{sn}' \oplus \mathcal{Q}_{sn_1} \oplus \cdots \oplus \mathcal{Q}_{sn_N}$.
Define
(b3) $\mathcal{Q}_{cm} = \mathcal{Q}_{cm}' \oplus \mathcal{Q}_{cm_1} \oplus \cdots \oplus \mathcal{Q}_{cm_N}$,
where $\Gamma \vdash \mathcal{Q}_{sn} \sim_{\mathcal{U}} \mathcal{Q}_{cm}$.
By $\mathcal{E}_\alpha$,
(b4) $\forall i \in [1, N]$, $\Gamma \vdash \mathcal{Q}_{sn_i} \sim_{\mathcal{U}} \mathcal{Q}_{cm_i}$.
Using (b4) define
(b5) $\mathcal{C}_{cm}' \triangleq \mathcal{Q}_{cm}' \rhd (\mathcal{S}_{cm_1} \circ \mathcal{Q}_{cm_1}) \cdots (\mathcal{S}_{cm_N} \circ \mathcal{Q}_{cm_N})$.
Using (b5) apply CM-DeQueue and obtain
(b6) $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$
By $\mathcal{E}_\alpha$, $\mathcal{E}_\beta$, (b4), $\mathcal{E}_\gamma$, $\mathcal{E}_\delta$, and $\mathcal{E}_\epsilon$,
(b7) $\mathcal{C}_{sn}' \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{C}_{cm}'$.
By (b6) and (b7),
The conclusion holds.

□

Lemma 8 (Single-step transition per node: semi-naïve simulates online compression)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N} \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots, \mathcal{S}_{cm_N}$
and $\mathcal{S}_{sn_\ell} \hookrightarrow \mathcal{S}_{sn_\ell}', \mathcal{U}_{sn_{ext}}'$
implies
$\exists \mathcal{U}_{cm_{ext}}'$, $\exists \mathcal{S}_{cm_\ell}'$ *s.t.*
$\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm_\ell}', \mathcal{U}_{cm_{ext}}'$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}' \cdots \mathcal{S}_{sn_N} \ \boldsymbol{\mathcal{R}_{\mathcal{C}}} \ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}' \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}' \cdots \mathcal{S}_{cm_N}$.

Proof.

Assume
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell} \cdots, \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN}$
(2) $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn'_\ell}, \mathcal{U}_{sn'_{ext}}$
By inversion on (1) we have
$\forall i \in [1,N]$, $\mathcal{S}_{sni} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sni}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1,N]$, $\mathcal{S}_{cmi} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cmi}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}$ $\boldsymbol{\mathcal{R}_U}$ $\mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1,N]$, $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sni}$ $\boldsymbol{\mathcal{R}_U}$ $\bigcup_{i=1}^{N} \mathcal{U}_{cmi}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cmi}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i$ $\boldsymbol{\mathcal{R}_{re}}$ $\bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$ $\boldsymbol{\mathcal{R}_{prov}}$ $\bigcup_{i=1}^{N} \Upsilon_{prov_i}$
We proceed by induction over the rules for (2)

Case A: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn'_\ell}, \mathcal{U}_{sn'_{ext}}$ was SN-Event**.**
By inversion we know:
(a1) $\mathcal{S}_{sn\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, e(@\iota_\ell, \vec{t_\ell}) :: \mathcal{E}_\ell, \mathcal{U}_{sn\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
(a2) $\mathcal{S}_{sn'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{sn\ell} \circ [u_{sn\ell}], equiSet'_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle, \mathcal{U}_{sn'_{ext}}$
(a3) $u_{sn\ell} = e(@\iota_e, \vec{t_e})$
(a4) $\Gamma(e)[tuple] = \mathsf{event}$
(a5) $K = \Gamma(e)[equi\_attr]$
(a6) $heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), K)$
(a7) $equiSet' = equiSet \cup heq$
We define
(a8) $u_{cm\ell} \triangleq \langle e(@\iota_\ell, \vec{t_e}), \mathsf{createFlag}, \mathtt{eID}, heq \rangle$
where $\mathsf{createFlag} = NCreate$ if $heq \in equiSet_\ell$ and $\mathsf{createFlag} = Create$ if $heq \notin equiSet_\ell$
(a9) $\mathtt{eID} = \textsc{TupleHash}(e(@\iota_e, \vec{t_e}), \Gamma)$
By the definition of $u_{cm\ell}$,
(a10) $\Gamma \vdash u_{sn\ell} \sim_u u_{cm\ell}$.
By $\mathcal{E}_\beta$ and (a10),
(a11) $\Gamma \vdash \mathcal{U}_{sn\ell} :: u_{sn\ell} \sim_{\mathcal{U}} \mathcal{U}_{cm\ell} :: u_{cm\ell}$
By (a11) we apply CM-Init-Event to obtain
(a12) $\mathcal{S}_{cm\ell} \hookrightarrow \mathcal{S}_{cm'_\ell}, []$
where $\mathcal{S}_{cm'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{sn\ell} \circ [u_{sn\ell}], equiSet'_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$.
By $\mathcal{E}_\alpha$, $\mathcal{E}_\beta$, $\mathcal{E}_\gamma$, $\mathcal{E}_\delta$, $\mathcal{E}_\epsilon$ and (a11),
(a13) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn'_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cmN}$
By (a12) and (a13),
the conclusion holds

Case B: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn'_\ell}, \mathcal{U}_{sn'_{ext}}$ was SN-RuleFire-Fast**.**
By inversion we know:
(b1) $\mathcal{S}_{sn\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
(b2) $\mathcal{S}_{sn'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn\ell} \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
(b3) $\Gamma(q)[tuple] = \mathsf{fast}$
(b4) $u_{sn\ell} = tr_q{:}q(@\iota_\ell, \vec{t_q})$
(b5) $fireRulesSN(@\iota_\ell, \Delta DQ, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$.
By (1), the above and since $DQ \subseteq DQ$, we apply *fireRulesSN* simulates *fireRulesCM* (Lemma 10) to obtain that
(b6) $\exists \mathcal{U}_{cm'_{\ell,in}}$, $\exists \mathcal{U}_{cm'_{\ell,ext}}$, $\exists \Upsilon'_\ell$ s.t.
$fireRulesCM(@\iota_\ell, \Delta\bar{DQ}, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{\ell,ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn^\partial_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{\ell,ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm^\partial_\ell} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn^\partial_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn\ell} \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm^\partial_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.
We apply CM-RuleFire-Fast to obtain
(b7) $\mathcal{S}_{cm\ell} \hookrightarrow \mathcal{S}_{cm^\partial_\ell}, \mathcal{U}_{cm'_{\ell,ext}}$
By (b6) and (b7)
the conclusion holds

Case C: The last rule that derived $\mathcal{S}_{sn\ell} \hookrightarrow \mathcal{S}_{sn'_\ell}, \mathcal{U}_{sn'_{ext}}$ was SN-RuleFire-Interest**.**
By inversion we know
(c1) $\mathcal{S}_{sn\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
(c2) $\mathcal{S}_{sn'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell \cup p(@\iota_\ell, \vec{t_p}), \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn\ell} \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \cup \{\mathsf{interest}(tr_p{:}p(@\iota_\ell, \vec{t_p}))\} \rangle, \mathcal{U}_{sn'_{ext}}$
(c3) $\Gamma(p)[tuple] = \mathsf{interest}$
(c4) $u_{sn\ell} = tr_p{:}p(@\iota_\ell, \vec{t_p})$
(c5) $fireRulesSN(@\iota_\ell, \Delta DQ, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$
By (1) and since $DQ \subseteq DQ$ we apply *fireRulesSN* simulates *fireRulesCM* (Lemma 10) to obtain

(c6) $\exists \mathcal{U}_{cm'_{\ell,in}}$, $\exists \mathcal{U}_{cm'_{\ell,ext}}$, $\exists \Upsilon'_\ell$ s.t.
$\mathit{fireRulesCM}(@\iota_\ell, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{\ell,in}}, \mathcal{U}_{cm'_{\ell,ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{\ell,ext}} \rhd \mathcal{S}_{sn1}, \cdots, \mathcal{S}_{sn_\ell}^{\partial}, \cdots, \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{\ell,ext}} \rhd \mathcal{S}_{cm1}, \cdots, \mathcal{S}_{cm_\ell}^{\partial}, \cdots, \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn_\ell} \circ \mathcal{U}_{sn'_{in}}, \mathit{equiSet}_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell} \circ \mathcal{U}_{cm'_{in}}, \mathit{equiSet}_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.

Using the above constructs we define

(c7) $\texttt{tID} \triangleq \textsc{TupleHash}(p(@\iota_\ell, \vec{t}_p), K)$

(c8) $\mathit{prov} \triangleq \langle @\iota_p, \texttt{tID}, \texttt{eID}, \lambda_p \rangle$.

By examining the rules for Semi-Naïve Evaluation,

$tr_p \in \mathcal{M}_\ell$

By the above and $\mathcal{E}_\delta$,

(c9) $\exists yl_p$, $\exists \lambda_q$ s.t.
$\Gamma \vdash tr_p \sim_d yl_p$
and $tl(yl){:}1 = \lambda_p$
and $\Gamma, \Upsilon'_\ell \cup \bigcup_{i=1}^{N} \Upsilon_i \vdash \mathsf{interest}(tr) \sim_{prov} \mathit{prov}$.

By (c9) and $\mathcal{E}_\epsilon$

(c10) $\Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_{prov_i} \cup \mathcal{M}_{prov'_\ell}$ $\boldsymbol{\mathcal{R}_{prov}}$ $\bigcup_{i=1, i\neq\ell}^{N} \Upsilon_{prov_i} \cup (\Upsilon_{prov_\ell} \cup \mathit{prov})$

By $\mathcal{E}_\beta$,

(c11) $\forall i \in [1, N]$, $\Gamma \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{sn_i} \cup \bar{\mathcal{U}}_{sn_\ell}$ $\boldsymbol{\mathcal{R}_\mathcal{U}}$ $\bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup \bar{\mathcal{U}}_{cm_\ell}$

Apply CM-RuleFire-Interest to obtain

(c12) $\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm'_\ell}, []$,
where $\mathcal{S}_{cm'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell \cup p(@\iota_\ell, \vec{t}_p), \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \cup \mathit{prov} \rangle$.

We apply Deleting updates that triggered all possible rules (Lemma 9) to obtain:

By $\mathcal{E}_\alpha$, (c11), $\mathcal{E}_\gamma$, $\mathcal{E}_\delta$, and (c10),

(c13) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn'_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cmN}$

By (c12) and (c13),

the conclusion holds

□

Lemma 9 (Deleting updates that triggered all possible rules)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $u_{sn_\ell} = tr_q{:}Q$
and $\forall r \in DQ$,
$r = \mathit{rID}\ p(@\ell_p, \vec{x}_p) \coloneq q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
$\Sigma' = \rho(\Delta r, Q, \mathcal{DB}_\ell)$
and $\Sigma = \mathit{sel}(\Sigma', \Delta r)$
implies
$\forall \sigma \in \Sigma$, $(\mathit{rID}, p(@\ell_p, \vec{x}_p), tr_q{:}Q, b_1(@\ell_q, \vec{x}_{b1}) :: \cdots :: b_n(@\ell_q, \vec{x}_{bn}))\sigma \in \bigcup_{i=1}^{N} \mathcal{M}_i$
implies
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn'_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn_\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm'_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.
Assume that:

(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$

(2) $u_{sn_\ell} = tr_q{:}Q$

(3) $\forall r \in DQ$,
$r = \mathit{rID}\ p(@\ell_p, \vec{x}_p) \coloneq q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
$\Sigma' = \rho(\Delta r, Q, \mathcal{DB}_\ell)$
and $\Sigma = \mathit{sel}(\Sigma', \Delta r)$
implies
$\forall \sigma \in \Sigma$, $(\mathit{rID}, p(@\ell_p, \vec{x}_p), tr_q{:}Q, b_1(@\ell_q, \vec{x}_{b1}) :: \cdots :: b_n(@\ell_q, \vec{x}_{bn}))\sigma \in \bigcup_{i=1}^{N} \mathcal{M}_i$

By inversion on the rules that derive (1),

$\forall i \in [1,N]$, $\mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i}\rangle$
$\forall i \in [1,N]$, $\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i}\rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1,N],\ \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{\mathbf{prov}}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$

Case A: $u_{cm_\ell} \notin \mathcal{U}_{cm}{}^F$
By $\mathcal{E}_\beta$,
(a1) $\forall i \in [1,N]\backslash\ell$, $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$ and $\Gamma \vdash \bar{\mathcal{U}}_{sn_\ell}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bar{\mathcal{U}}_{cm_\ell}$
By $\mathcal{E}_\alpha$, (a1), $\mathcal{E}_\gamma$, $\mathcal{E}_\delta$, and $\mathcal{E}_\epsilon$,
The conclusion holds

Case B: $u_{cm_\ell} \in \mathcal{U}_{cm}{}^F$
By $\mathcal{E}_\delta$,
(b1) $\exists ruleExec_p$ s.t. $DQ, \Gamma \vdash u_{cm_\ell} \twoheadrightarrow ruleExec_p$
By (b1),
(b2) $\exists r \in DQ$ s.t.
$r = rID\ p(@\ell_p, \vec{x}_p) \coloneq q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
and $\Sigma' = \rho(\Delta r, Q, \mathcal{DB}_\ell)$
and $\Sigma = sel(\Sigma', \Delta r)$
and $\exists \sigma \in \Sigma$, $\exists yl_q$ s.t.
$\Gamma \vdash (rID, p(@\ell_p, \vec{x}_p), tr_q{:}Q, b_1(@\ell_q, \vec{x}_{b1}) :: \cdots :: b_n(@\ell_q, \vec{x}_{bn}))\sigma \sim\sim_d yl_q :: ruleExec_p$

Subcase I: $u_{cm_\ell}.\mathsf{createFlag} = Create$
By the assumption,
(i1) $ruleExec_p \in \bigcup_{i=1}^{N} \Upsilon_i$
By (i1),
(i2) $yl_q :: ruleExec_p \subseteq \bigcup_{i=1}^{N} \Upsilon_i$
Thus we can define
(i3) $\mathcal{U}_{cm}{}^{F'} \triangleq \mathcal{Q}_{cm} \cup \bigcup_{i=1, i\neq\ell} \mathcal{U}_{cm_i} \cup \bar{\mathcal{U}}_{cm_\ell}$

By inversion on $\mathcal{E}_\delta$, we have
(i4) $\Gamma, DQ \vdash \mathcal{U}_{cm}{}^F \twoheadrightarrow \Upsilon^F$
(i5) $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{M}_i \approx_d \bigcup_{i=1}^{N} \Upsilon_i \cup \Upsilon^F$
By (i3) and (i4),
(i6) $\Gamma, DQ \vdash \mathcal{U}_{cm}{}^{F'} \twoheadrightarrow \Upsilon^{F'}$ where $ruleExec_p \notin \Upsilon^{F'}$
By (i1), (i5), and (i6),
(i7) $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{M}_i \approx_d \bigcup_{i=1}^{N} \Upsilon_i \cup \Upsilon^{F'}$
where $ruleExec_p \in \bigcup_{i=1}^{N} \Upsilon_i \cup \Upsilon^{F'}$
By (i6) and (i7),
(i8) $\Gamma, DQ, \mathcal{U}_{cm}{}^{F'} \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$

By (i8) and $\mathcal{E}_\epsilon$,
$\mathcal{E}'_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^{F'}, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{\mathbf{prov}}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$

By $\mathcal{E}_\alpha$, $\mathcal{E}_\beta$, (i8), $\mathcal{E}'_\epsilon$, $\mathcal{E}'_\delta$
the conclusion follows

Subcase II: $u_{cm_\ell}.\mathsf{createFlag} = NCreate$
We claim that $u_{cm_\ell} \notin \mathcal{U}_{cm}{}^F$, a contradiction
This follows by definition of the relations that generate $\Gamma \vdash u_{cm}{}^F \twoheadrightarrow \Upsilon^F$, as
$\forall u_{cm} \in \mathcal{U}_{cm}{}^F$, $u_{cm}.\mathsf{createFlag} = Create$

□

Lemma 10 (*fireRulesSN* simulates *fireRulesCM*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell}\rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell}\rangle$
and $\bar{DQ} \subseteq DQ$
and $fireRulesSN(@\iota_\ell, \Delta\bar{DQ}, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$
implies

$\exists \mathcal{U}_{cm'_{in}}$, $\exists \mathcal{U}_{cm'_{ext}}$, $\exists \Upsilon'_\ell$ *s.t.*
fireRulesCM$(@\iota_\ell, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.

Assume that
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
(2) $\bar{DQ} \subseteq DQ$
(3) *fireRulesSN*$(@\iota_\ell, \Delta\bar{DQ}, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$
By inversion on the rules for (1),
$\forall i \in [1, N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1, N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}$ $\boldsymbol{\mathcal{R}_U}$ $\mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N], \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}$ $\boldsymbol{\mathcal{R}_U}$ $\bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i$ $\boldsymbol{\mathcal{R}_{re}}$ $\bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$ $\boldsymbol{\mathcal{R}_{prov}}$ $\bigcup_{i=1}^{N} \Upsilon_{prov_i}$.
We proceed by induction over $|\Delta\bar{DQ}|$.

Base Case: $|\Delta\bar{DQ}| = 0$**.**
By assumption,
(b1) $\Delta\bar{DQ} = []$.
By (b1)
the last rule that derived (3) is SN-Empty,
By inversion on SN-Empty
(b2) $\mathcal{U}_{sn'_{in}} = []$
(b3) $\mathcal{U}_{sn'_{ext}} = []$
(b4) $\mathcal{M}'_\ell = \mathcal{M}_\ell$
Using CM-Empty we have
(b5) *fireRulesCM*$(@\iota_\ell, [], u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = ([], [], \Upsilon_\ell)$
By (b2), (b3), (b4), and (1),
(b6) $\mathcal{Q}_{sn} \circ [] \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ [] \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cmN}$.
By (b5) and (b6),
the conclusion holds

Inductive Case: $|\Delta\bar{DQ}| = k + 1$**.**
By assumption
the last rule that derived (3) is SN-Seq
By inversion we have
(i1) *fireSingleRuleSN*$(@\iota_\ell, \Delta r, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn^1_{in}}, \mathcal{U}_{sn^1_{ext}}, \mathcal{M}^1_\ell)$
(i2) *fireRulesSN*$(@\iota_\ell, \Delta\hat{DQ}, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}^1_\ell) = (\mathcal{U}_{sn^2_{in}}, \mathcal{U}_{sn^2_{ext}}, \mathcal{M}'_\ell)$
where $\Delta\bar{DQ} = \Delta r :: \Delta\hat{DQ}$
and $\mathcal{U}_{sn'_{in}} = \mathcal{U}_{sn^1_{in}} \circ \mathcal{U}_{sn^2_{in}}$
and $\mathcal{U}_{sn'_{ext}} = \mathcal{U}_{sn^1_{ext}} \circ \mathcal{U}_{sn^2_{ext}}$.
Since $r \in DQ$, we apply *fireSingleRuleSN* simulates *fireSingleRuleCM* (Lemma 11) to obtain:
(i3) $\exists \mathcal{U}_{cm^1_{in}}$, $\exists \mathcal{U}_{cm^1_{ext}}$, $\exists \Upsilon^1_\ell$ s.t.
fireSingleRuleCM$(@\iota_\ell, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm^1_{in}}, \mathcal{U}_{cm^1_{ext}}, \Upsilon^1_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn^1_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial 1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm^1_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial 1} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn_\ell}^{\partial 1} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \mathcal{U}_{sn_\ell} \circ \mathcal{U}_{sn^1_{in}}, equiSet_\ell, \mathcal{M}^1_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial 1} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \mathcal{U}_{cm_\ell} \circ \mathcal{U}_{cm^1_{in}}, equiSet_\ell, \Upsilon^1_\ell, \Upsilon_{prov_\ell} \rangle$
Since $|\hat{DQ}| = k$ and $\hat{DQ} \subseteq DQ$, and using (i3) we apply the induction hypothesis to obtain
(i4) $\exists \mathcal{U}_{cm^2_{in}}$, $\exists \mathcal{U}_{cm^2_{ext}}$, $\exists \Upsilon'_\ell$ s.t.
fireRulesCM$(@\iota_\ell, \Delta\hat{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon^1_\ell) = (\mathcal{U}_{cm^2_{in}}, \mathcal{U}_{cm^2_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn^1_{ext}} \circ \mathcal{U}_{sn^2_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial 2} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{sn^1_{ext}} \circ \mathcal{U}_{cm^2_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial 2} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell}^{\partial 2} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn_\ell} \circ \mathcal{U}_{cm^1_{in}} \circ \mathcal{U}_{sn^2_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial 2} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell} \circ \mathcal{U}_{cm^1_{in}} \circ \mathcal{U}_{cm^2_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.

By the above we apply CM-Seq to obtain
(i5) $\mathit{fireRulesCM}(@\iota_\ell, \Delta\bar{D}Q, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm^1_{in}} \circ \mathcal{U}_{cm^2_{in}}, \mathcal{U}_{cm^1_{ext}} \circ \mathcal{U}_{cm^2_{ext}}, \Upsilon'_\ell)$
where $\mathcal{S}_{sn^\partial_\ell} \triangleq \mathcal{S}_{sn^{\partial 2}_\ell}$
and $\mathcal{S}_{cm^\partial_\ell} \triangleq \mathcal{S}_{cm^{\partial 2}_\ell}$
By (i4) and (i5),
the conclusion holds.

□

Lemma 11 (*fireSingleRuleSN* simulates *fireSingleRuleCM*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN}$
and $r \in DQ$
and $\mathcal{S}_{sn\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov\ell}\rangle$
and $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov\ell}\rangle$
and $\mathit{fireSingleRuleSN}(@\iota_\ell, \Delta r, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn^\partial_{in}}, \mathcal{U}_{sn^\partial_{ext}}, \mathcal{M}^\partial_\ell)$
implies
$\exists \mathcal{U}_{cm^\partial_{in}},\ \exists \mathcal{U}_{cm^\partial_{ext}},\ \exists \Upsilon^\partial_\ell$ *s.t.*
$\mathit{fireSingleRuleCM}(@\iota_\ell, \Delta r, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm^\partial_{in}}, \mathcal{U}_{cm^\partial_{ext}}, \Upsilon^\partial_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn^\partial_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn^\partial_\ell} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm^\partial_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm^\partial_\ell} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn^\partial_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ \mathcal{U}_{sn^\partial_{in}}, \mathit{equiSet}_\ell, \mathcal{M}^\partial_\ell, \mathcal{M}_{prov\ell}\rangle$
and $\mathcal{S}_{cm^\partial_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ \mathcal{U}_{cm^\partial_{in}}, \mathit{equiSet}_\ell, \Upsilon^\partial_\ell, \Upsilon_{prov\ell}\rangle$.

Proof.
Assume
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN}$
(2) $r \in DQ$
(3) $\mathcal{S}_{sn\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov\ell}\rangle$
(4) $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov\ell}\rangle$
(5) $\mathit{fireSingleRuleSN}(@\iota_\ell, \Delta r, u_{sn\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn^\partial_{in}}, \mathcal{U}_{sn^\partial_{ext}}, \mathcal{M}^\partial_\ell)$
By inversion on the rules that derive (1),
$\forall i \in [1, N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, \mathit{equiSet}_i, \mathcal{M}_i, \mathcal{M}_{prov_i}\rangle$
$\forall i \in [1, N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, \mathit{equiSet}_i, \Upsilon_i, \Upsilon_{prov_i}\rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_U}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N]\backslash\ell,\ \Gamma \vdash \bigcup_{i=1}^N \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_U}\ \bigcup_{i=1}^N \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm^F} \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^N \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm^F} \vdash \bigcup_{i=1}^N \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{re}}\ \bigcup_{i=1}^N \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm^F}, \bigcup_{i=1}^N \Upsilon_i \vdash \bigcup_{i=1}^N \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{prov}}\ \bigcup_{i=1}^N \Upsilon_{prov_i}$.
By examining the rules for *fireSingleRuleSN*, the last rule that derived (5) was SM-FireSingle. By inversion we have:
(6) $\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \coloneq \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(7) $u_{sn\ell} = tr_q{:}q(@\iota_q, \vec{t_q})$
(8) $\bar{\Sigma} = \rho(\Delta r, q(@\iota_q, \vec{t_q}), \mathcal{DB}_\ell)$
(9) $\Sigma = \mathit{sel}(\bar{\Sigma}, \Delta r)$
(10) $\mathit{derivationSN}(@\iota_q, \Sigma, \Delta r, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn^\partial_{in}}, \mathcal{U}_{sn^\partial_{ext}}, \mathcal{M}^\partial_\ell)$
Given $\Sigma \subseteq \Sigma = \mathit{sel}(\bar{\Sigma}, \Delta r)$, using the above, we apply *derivationSN* simulates *compressionCM* (Lemma 12) to obtain:
(11) $\exists \mathcal{U}_{cm'_{in}},\ \exists \mathcal{U}_{cm'_{ext}},\ \exists \Upsilon'_\ell$ s.t.
$\mathit{compressionCM}(@\iota_\ell, \Sigma^\partial, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm^\partial_{in}}, \mathcal{U}_{cm^\partial_{ext}}, \Upsilon^\partial_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn'_\ell} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cmN}$
By the above we apply CM-FireSingle to obtain
(12) $\mathit{fireSingleRuleCM}(@\iota_\ell, \Delta r, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}^\partial_{in}, \mathcal{U}_{cm^\partial_{ext}}, \Upsilon^\partial_\ell)$
By (11) and (12)
the conclusion holds

□

Lemma 12 (*derivationSN* simulates *compressionCM*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1}, \cdots, \mathcal{S}_{sn\ell}, \cdots, \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1}, \cdots, \mathcal{S}_{cm\ell}, \cdots, \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, \mathit{equiSet}_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov\ell}\rangle$
and $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, \mathit{equiSet}_\ell, \Upsilon_\ell, \Upsilon_{prov\ell}\rangle$
and $r \in DQ$
and $u_{sn\ell} = tr_q{:}q(@\iota_\ell, \vec{t_q})$
and $\bar{\Sigma} = \rho(\Delta r, q(@\iota_\ell, \vec{t_q}), \mathcal{DB}_\ell)$
and $\Sigma \subseteq \mathit{sel}(\bar{\Sigma}, \Delta r)$
$\mathit{derivationSN}(@\iota, \Sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn^\partial_{in}}, \mathcal{U}_{sn^\partial_{ext}}, \mathcal{M}^\partial_\ell)$

implies
$\exists \mathcal{U}_{cm_{in}}^{\partial}$, $\exists \mathcal{U}_{cm_{ext}}^{\partial}$, $\exists \Upsilon_{\ell}^{\partial}$ *s.t.*
$compressionCM(@_{\iota_\ell}, \Sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}^{\partial}, \mathcal{U}_{cm_{ext}}^{\partial}, \Upsilon_\ell^{\partial})$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{\partial} \rhd \mathcal{S}_{sn1}, \cdots, \mathcal{S}_{sn_\ell}^{\partial}, \cdots, \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}^{\partial} \rhd \mathcal{S}_{cm1}, \cdots, \mathcal{S}_{cm_\ell}^{\partial}, \cdots, \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn_{in}}^{\partial}, equiSet_\ell, \mathcal{M}_\ell^{\partial}, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm_{in}}^{\partial}, equiSet_\ell, \Upsilon_\ell^{\partial}, \Upsilon_{prov_\ell} \rangle$.

Proof.
Assume that
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell} = \langle @_{\iota_q}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @_{\iota_q}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
(2) $r \in DQ$
(3) $u_{sn_\ell} = tr_q{:}q(@_{\iota_\ell}, \vec{t}_q)$
(4) $\bar{\Sigma} = \rho(\Delta r, q(@_{\iota_\ell}, \vec{t}_q), \mathcal{DB}_\ell)$
(5) $\Sigma \subseteq sel(\bar{\Sigma}, \Delta r)$
(6) $derivationSN(@_\iota, \Sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}^{\partial}, \mathcal{U}_{sn_{ext}}^{\partial}, \mathcal{M}_\ell^{\partial})$
We proceed by induction over the length of Σ.

Base Case: $|\Sigma| = 0$**.**
By assumption,
(b1) $\Sigma = []$
We apply CM-Compress-Empty to obtain:
(b2) $compressionCM(@_{\iota_\ell}, [], \Delta r, u_{cm_\ell}, \Upsilon_\ell) = ([], [], \Upsilon'_\ell)$
where $\Upsilon'_\ell = \Upsilon_\ell$
Using the above, we define
(b3) $\mathcal{U}_{cm_{in}}^{\partial} \triangleq []$
(b4) $\mathcal{U}_{cm_{ext}}^{\partial} \triangleq []$
(b5) $\Upsilon_\ell^{\partial} \triangleq \Upsilon_\ell$
(b6) $\mathcal{S}_{sn_\ell}^{\partial} \triangleq \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
(b7) $\mathcal{S}_{cm_\ell}^{\partial} \triangleq \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.
By the above,
(b8) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{\partial} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{snN} = \mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$
(b9) $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}^{\partial} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cmN} = \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cmN}$
By the above constructs,
(b10) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{\partial} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}^{\partial} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cmN}$
By (b2) and (b10),
the conclusion holds

Inductive Case: $|\Sigma| = k + 1$**.**
By assumption,
(i1) The last transition rule that derived $derivationSN(@_\iota, \Sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}^{\partial}, \mathcal{U}_{sn_{ext}}^{\partial}, \mathcal{M}_\ell^{\partial})$ was SN-Subst
By inversion on SN-Subst,
(i2) $\exists \sigma$, $\exists \hat{\Sigma}$ s.t. $\Sigma = \sigma :: \hat{\Sigma}$
(i3) $\exists \mathcal{U}_{sn_{in}}^{1}$, $\exists \mathcal{U}_{sn_{in}}^{2}$, $\exists \mathcal{U}_{sn_{ext}}^{1}$, $\exists \mathcal{U}_{sn_{ext}}^{2}$, $\exists \mathcal{M}_\ell^1$ s.t.
$singleDerivSN(@_{\iota_\ell}, \sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}^{1}, \mathcal{U}_{sn_{ext}}^{1}, \mathcal{M}_\ell^1)$
and $derivationSN(@_{\iota_\ell}, \hat{\Sigma}, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell^1) = (\mathcal{U}_{sn_{in}}^{2}, \mathcal{U}_{sn_{ext}}^{2}, \mathcal{M}_\ell^{\partial})$
We apply *singleDerivSN* simulates *singleCompressionCM* (Lemma 13) to obtain that
(i4) $\exists \mathcal{U}_{cm_{in}}^{1}$, $\exists \mathcal{U}_{cm_{ext}}^{1}$, $\exists \Upsilon_\ell^1$ s.t.
$singleCompressionCM(@_{\iota_\ell}, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}^{1}, \mathcal{U}_{cm_{ext}}^{1}, \Upsilon_\ell^1)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell}^{\partial 1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial 1} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell}^{\partial 1} = \langle @_{\iota_q}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn_{in}}^{1}, equiSet_\ell, \mathcal{M}_\ell^1, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial 1} = \langle @_{\iota_q}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm_{in}}^{1}, equiSet_\ell, \Upsilon_\ell^1, \Upsilon_{prov_\ell} \rangle$
Since $\Sigma = \sigma :: \hat{\Sigma}$ and $|\Sigma| = k + 1$, thus $|\hat{\Sigma}| = k$.
Using the above constructs we apply I.H. to find that
(i5) $\exists \mathcal{U}_{cm_{in}}^{2}$, $\exists \mathcal{U}_{cm_{ext}}^{2}$, $\exists \Upsilon_\ell^{\partial}$ s.t.
$compressionCM(@_{\iota_\ell}, \hat{\Sigma}, \Delta r, u_{cm_\ell}, \Upsilon_\ell^1) = (\mathcal{U}_{cm_{in}}^{2}, \mathcal{U}_{cm_{ext}}^{2}, \Upsilon_\ell^{\partial})$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{1} \circ \mathcal{U}_{sn_{ext}}^{2} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}^{1} \circ \mathcal{U}_{cm_{ext}}^{2} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cmN}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn_{in}}^{1} \circ \mathcal{U}_{sn_{in}}^{2}, equiSet_\ell, \mathcal{M}_\ell^{\partial}, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @_{\iota_\ell}, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm_{in}}^{1} \circ \mathcal{U}_{cm_{in}}^{2}, equiSet_\ell, \Upsilon_\ell^{\partial}, \Upsilon_{prov_\ell} \rangle$
Applying CM-Subst we have
(i6) $derivationSN(@_{\iota_q}, \Sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}^{\partial}, \mathcal{U}_{cm_{ext}}^{\partial}, \Upsilon_\ell^{\partial})$

where $\Sigma = \sigma :: \hat{\Sigma}$
and $\mathcal{U}_{cm_{in}}^{\partial} = \mathcal{U}_{cm_{in}}^{1} \circ \mathcal{U}_{cm_{in}}^{2}$
and $\mathcal{U}_{cm_{ext}}^{\partial} = \mathcal{U}_{cm_{ext}}^{1} \circ \mathcal{U}_{cm_{ext}}^{2}$

By (i5) and (i6)

The conclusion holds.

□

Lemma 13 (*singleDerivSN* simulates *singleCompressionCM*.)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$,
where $\mathcal{S}_{sn_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $\bar{\Sigma} = \rho(\Delta r, q(@\iota_q, \vec{t}_\ell), \mathcal{DB}_\ell)$
and $\sigma \in sel(\bar{\Sigma}, \Delta r)$
and $singleDerivSN(@\iota_\ell, \sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}^{\partial}, \mathcal{U}_{sn_{ext}}^{\partial}, \mathcal{M}_\ell^{\partial})$
implies
$\exists \mathcal{U}_{cm_{in}}^{\partial}$, $\exists \mathcal{U}_{cm_{ext}}^{\partial}$, $\exists \Upsilon_\ell^{\partial}$ *s.t.*
$singleCompressionCM(@\iota_\ell, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}^{\partial}, \mathcal{U}_{cm_{ext}}^{\partial}, \Upsilon'_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{\partial} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}^{\partial} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn_{in}}^{\partial}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm_{in}}^{\partial}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.
Assume the following:
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$,
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
(2) $r \in DQ$
(3) $\bar{\Sigma} = \rho(\Delta r, q(@\iota_q, \vec{t}_\ell), \mathcal{DB}_\ell)$
(4) $\sigma \in sel(\bar{\Sigma}, \Delta r)$
(5) $singleDerivSN(@\iota_\ell, \sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}^{\partial}, \mathcal{U}_{sn_{ext}}^{\partial}, \mathcal{M}_\ell^{\partial})$

Then by inversion on the rules for $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$, we have
$\forall i \in [1, N]$, $\mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1, N]$, $\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_U}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N], \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_U}\ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{\mathbf{prov}}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$.

By inversion on the rules for $\mathcal{E}_\delta$, we have
$\mathcal{E}_1 :: \Gamma, DQ \vdash \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i} \curvearrowright \Upsilon^F$
$\mathcal{E}_2 :: \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{M}_i \approx_d \bigcup_{i=1}^{N} \Upsilon_i \cup \Upsilon^F$.

Case I: $\Gamma(q)[tuple] = \mathsf{event}$**.**
The last transition rule that derived (5) was SN-SingleSubst, thus by inversion we have:
(1) $\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(2) $u_{sn_\ell} = q(@\iota_q, \vec{t}_q)$
(3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q)$
(4) $\Gamma(q)[type] = \mathsf{event}$
(5) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{N} \vec{x}_{bi}$
(6) $tr_p = (rID, p(@\ell_p, \vec{x}_p)\sigma, q(@\iota_q, \vec{t}_q), b_1(@\ell_q, \vec{x}_{b1}))\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$
(7) $u_{sn'_\ell} = tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$
(8) if $(\sigma(@\ell_p) = @\iota_q)$ then $\mathcal{U}_{sn'_{in}} = [u_{sn'_\ell}], \mathcal{U}_{sn'_{ext}} = []$ else $\mathcal{U}_{sn'_{in}} = [], \mathcal{U}_{sn'_{ext}} = [u_{sn'_\ell}]$
(9) $\mathcal{M}'_\ell = \mathcal{M}_\ell \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

Subcase A: $u_{cm_\ell}.\mathsf{createFlag} = Create$**.**
By $\mathcal{E}_\beta$ we have
(a1) $\Gamma \vdash u_{sn_\ell} \sim_u u_{cm_\ell}$
By assumption,
(a2) the last rule the derived (a1) was u-Base

By the above and inversion we have:
(a3) $heq = \textsc{EquiHash}(q(@\iota_q, \vec{t_q}), \Gamma)$
(a4) $\texttt{eID} = \textsc{TupleHash}(q(@\iota_q, \vec{t_q}), \Gamma)$

Using the above constructs we define the following:
(a5) $\forall i \in [1,n]$, $\texttt{vID}_i \triangleq \mathsf{hash}(b_i(@\ell_q, \vec{x}_{bi})\sigma)$
(a6) $ruleargs_p \triangleq rID :: \iota_q :: \texttt{vID}_1 :: \cdots :: \texttt{vID}_n$
(a7) $\texttt{HrID}_p \triangleq \mathsf{hash}(ruleargs_p)$
(a8) $\lambda_q \triangleq \mathsf{id}(\emptyset, \emptyset, heq)$
(a9) $\lambda_p \triangleq \mathsf{id}(@\iota_q, \texttt{HrID}_p, heq)$
(a10) $ruleExec_p \triangleq \langle \lambda_p, ruleargs, \lambda_q \rangle$
(a11) $u_{cm}'_\ell \triangleq \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \texttt{eID}, \lambda_p \rangle$.
(a12) If $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{cm}{}^{\partial}_{in} \triangleq [u_{cm}'_\ell]$ and $\mathcal{U}_{cm}{}^{\partial}_{ext} \triangleq [])$ else $(\mathcal{U}_{cm}{}^{\partial}_{in} \triangleq []$ and $\mathcal{U}_{cm}{}^{\partial}_{ext} \triangleq [u_{cm}'_\ell])$
(a13) $\Upsilon'_\ell = \Upsilon_\ell \cup ruleExec_p$

We use the above constructs to apply CM-Create to obtain
(a14) $singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm}{}^{\partial}_{in}, \mathcal{U}_{cm}{}^{\partial}_{ext}, \Upsilon'_\ell)$

By definition of the constructs above we have
$\Gamma \vdash u_{sn}'_\ell \sim_u u_{cm}'_\ell$
By $\mathcal{E}_\alpha$ and the above,
(a14) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn}{}^{\partial}_{in}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm}{}^{\partial}_{in}$
By $\mathcal{E}_\beta$ and (a14),
(a15) $\forall i \in [1,N] \backslash \ell$, $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sni} \cup ((u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ \mathcal{U}_{sn}{}^{\partial}_{in})\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{cmi} \cup ((u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ \mathcal{U}_{cm}{}^{\partial}_{in})$

By definition $ruleExec_p$ we have
$\Gamma \vdash tr_p \sim_d ruleExec_p$
By $\mathcal{E}_\delta$ and the above,
(a16) $\Gamma, DQ, \mathcal{U}_{cm}{}^{F} \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1, i\neq\ell}^{N} \Upsilon_i \cup \Upsilon'_\ell$.

Using (a14), (a15), $\mathcal{E}_\gamma$, (a16), and $\mathcal{E}_\epsilon$, we have
(a17) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn}{}^{\partial}_{ext} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn}{}^{\partial}_{\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm}{}'_{ext} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm}{}^{\partial}_{\ell} \cdots \mathcal{S}_{cm_N}$

By (a14) and (a17),
the conclusion holds

Subase B: $u_{cm\ell}.\mathsf{createFlag} = NCreate$**.**
By $\mathcal{E}_\beta$ we have
(a1) $\Gamma \vdash u_{sn\ell} \sim_u u_{cm\ell}$
By assumption,
(a2) the last rule the derived (a1) was u-Base
By the above and inversion we have:
(a3) $heq = \textsc{EquiHash}(q(@\iota_q, \vec{t_q}), \Gamma)$
(a4) $\texttt{eID} = \textsc{TupleHash}(q(@\iota_q, \vec{t_q}), \Gamma)$

We define the following:
(a5) $\forall i \in [1,n]$, $\texttt{vID}_i \triangleq \mathsf{hash}(b_i(@\ell_e, \vec{x}_{bi})\sigma)$
(a6) $ruleargs_p \triangleq rID :: \iota_q :: \texttt{vID}_1 :: \cdots :: \texttt{vID}_n$
(a7) $\texttt{HrID}_p \triangleq \mathsf{hash}(ruleargs_p)$
(a9) $\lambda_p \triangleq \mathsf{id}(@\iota_q, \texttt{HrID}_p, heq)$
(a10) $ruleExec_p \triangleq \langle \lambda_p, ruleargs_p, \mathsf{id}(\emptyset, \emptyset, heq) \rangle$
(a11) $u_{cm}'_\ell \triangleq \langle p(@\ell_p, \vec{x}_p)\sigma, NCreate, \texttt{eID}, \lambda_p \rangle$.
(a12) If $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{cm}{}^{\partial}_{in} \triangleq [u_{cm}'_\ell]$ and $\mathcal{U}_{cm}{}^{\partial}_{ext} \triangleq [])$ else $(\mathcal{U}_{cm}{}^{\partial}_{in} \triangleq []$ and $\mathcal{U}_{cm}{}^{\partial}_{ext} \triangleq [u_{cm}'_\ell])$
(a13) $\Upsilon'_\ell \triangleq \Upsilon_\ell$

Using the above definitions we apply CM-NCreate to obtain
(a14) $singleCompressionCM(@\iota_\ell, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm}{}^{\partial}_{in}, \mathcal{U}_{cm}{}^{\partial}_{ext}, \Upsilon'_\ell)$

By definition of the constructs above we have
$\Gamma \vdash u_{sn}'_\ell \sim_u u_{cm}'_\ell$
By $\mathcal{E}_\alpha$ and the above,

(a15) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{in}}^{\partial}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{in}}^{\partial}$

By $\mathcal{E}_\beta$ and the above,

(a16) $\forall i \in [1,N]\backslash\ell,\ \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i} \cup ((u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn_{in}}^{\partial})\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bigcup_{i=1,i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup ((u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm_{in}}^{\partial})$

If $ruleExec_p \in \Upsilon_\ell$:

By assumption,

$\Upsilon'_\ell = \Upsilon_\ell$

Since $\Gamma \vdash tr_p : p(@\iota_p, \vec{t}_p) \sim_d ruleExec_p$, by $\mathcal{E}_\delta$ and the assumption that $ruleExec_p \in \Upsilon_\ell$, therefore

$\mathcal{M}'_\ell = \mathcal{M}_\ell$

By the above we have

(a17) $\Gamma, DQ, \mathcal{U}_{cm}{}^{F} \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$

If $ruleExec_p \notin \Upsilon_\ell$:

By examining the rules,

rule CM-Init-Event was fired in the past

$\exists u_{cm}{}_\ell^{\mu} \in \mathcal{U}_{cm_\ell}$ s.t. $u_{cm}{}_\ell^{\mu} = u_{cm_\ell}[\mathsf{createFlag} \mapsto \mathit{Create}]$

By construction,

$\Gamma \vdash u_{cm}{}_\ell^{\mu} \rightsquigarrow ruleExec_p, u_{cm'_\ell}[\mathsf{createFlag} \mapsto \mathit{Create}]$

By the above and $\mathcal{E}_\delta$ thus

(a18) $\Gamma, DQ, \mathcal{U}_{cm}{}^{F} \cup u_{cm}{}_\ell^{\mu} \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$

By (a15), (a16) $\mathcal{E}_\gamma$, (a17) or (a18), and $\mathcal{E}_\epsilon$, we have

(a18) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}^{\partial} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cm_N}.$

By (a14) and (a18),

The conclusion follows.

Case II: $\Gamma(q)[tuple] = \mathsf{fast}$ **or** $\Gamma(q)[tuple] = \mathsf{interest}$**.**

The last transition rule that derived (5) was SN-SingleSubst, thus by inversion we have:

(1) $\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \coloneq \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$

(2) $u_{sn_\ell} = tr_q{:}q(@\iota_q, \vec{t}_q)$

(3) $q(@\ell_q, \vec{x}_e)\sigma = q(@\iota_q, \vec{t}_q)$

(4) either $\Gamma(q)[type] = \mathsf{fast}$ or $\Gamma(q)[type] = \mathsf{interest}$

(5) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{N} \vec{x}_{bi}$

(6) $tr_p = (rID, p(@\ell_p, \vec{x}_p)\sigma, tr_q{:}q(@\iota_q, \vec{t}_q), b_1(@\ell_q, \vec{x}_{b1}))\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$

(7) $u_{sn'_\ell} = tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

(8) if $(\sigma(@\ell_p) = @\iota_q)$ then $\mathcal{U}_{sn'_{in}} = [u_{sn'_\ell}], \mathcal{U}_{sn'_{ext}} = []$ else $\mathcal{U}_{sn'_{in}} = [], \mathcal{U}_{sn'_{ext}} = [u_{sn'_\ell}]$

(9) $\mathcal{M}'_\ell = \mathcal{M}_\ell \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

Subcase A: $u_{cm_\ell}.\mathsf{createFlag} = \mathit{Create}$**.**

By $\mathcal{E}_\beta$ we have

(a1) $\Gamma \vdash u_{sn_\ell} \sim_u u_{cm_\ell}$

By assumption,

(a2) the last rule the derived (a1) was u-Ind

By the above and inversion we have:

(a3) $\Gamma \vdash tr_q{:}q(@\iota_q, \vec{t}_q) \sim_u \langle q(@\iota_q, \vec{t}_q), \mathsf{createFlag}, \mathtt{eID}, \lambda_q \rangle$

We define the following:

(a4) $\forall i \in [1,n]$, $\mathtt{vID}_i \triangleq \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$

(a5) $ruleargs_p \triangleq rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$

(a6) $heq \triangleq \textsc{EquiHash}(q(@\iota_q, \vec{t}_q), \Gamma)$

(a7) $\mathtt{HrID}_p \triangleq \mathsf{hash}(ruleargs_p)$

(a6) $\flat_p \triangleq \mathsf{hash}(\lambda_q)$

(a7) $\lambda_p \triangleq \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$

(a8) $ruleExec_p \triangleq \langle \lambda_p, ruleargs, \lambda_q \rangle$

(a9) $u_{cm'_\ell} \triangleq \langle p(@\ell_p, \vec{x}_p)\sigma, \mathit{Create}, \mathtt{eID}, \lambda_p \rangle$.

(a10) If $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{cm_{in}}^{\partial} \triangleq [u_{cm'_\ell}]$ and $\mathcal{U}_{cm_{ext}}^{\partial} \triangleq [])$ else $(\mathcal{U}_{cm_{in}}^{\partial} \triangleq []$ and $\mathcal{U}_{cm_{ext}}^{\partial} \triangleq [u_{cm'_\ell}])$

(a11) $\Upsilon'_\ell = \Upsilon_\ell \cup ruleExec_p$

We apply CM-Create to obtain

(a12) $singleCompressionCM(@\iota_\ell, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}^{\partial}, \mathcal{U}_{cm_{ext}}^{\partial}, \Upsilon'_\ell)$.

By (a3) the definition of the constructs we have
(a13) $\Gamma \vdash \mathcal{U}_{sn'_\ell} \sim_u u_{cm'_\ell}$
By $\mathcal{E}_\alpha$ the definitions of $\mathcal{U}_{sn^\partial_{ext}}$ and $\mathcal{U}_{cm^\partial_{ext}}$
(a14) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn^\partial_{ext}} \; \boldsymbol{\mathcal{R}_\mathcal{U}} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm^\partial_{ext}}$
By $\mathcal{E}_\beta$ and the above
(a15) $\forall i \in [1,N]$, $\Gamma \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{U}_{sn_i} \cup ((u_{sn_\ell} :: \bar{\mathcal{U}_{sn_\ell}}) \circ \mathcal{U}_{sn^\partial_{in}}) \; \boldsymbol{\mathcal{R}_\mathcal{U}} \; \bigcup_{i=1,i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup ((u_{cm_\ell} :: \bar{\mathcal{U}_{cm_\ell}}) \circ \mathcal{U}_{cm^\partial_{in}})$

By examining the transition rules for Semi-Naïve Evaluation and since $\Gamma(q)[type] = \mathsf{fast}$ we have
$tr_q \in \mathcal{M}_\ell$
By $\mathcal{E}_\delta$ and (a15) and since $\mathsf{createFlag} = \mathit{Create}$,
$\exists yl_q \subseteq \bigcup_{i=1}^{N} \mathcal{M}_i$ s.t. $\Gamma \vdash tr_q \sim_d yl_q$
By definition of tr_p and $ruleExec_p$ and the above,
$\Gamma \vdash tr_p \sim_d yl_q :: ruleExec_p$
By the above and $\mathcal{E}_\delta$ and since $ruleExec_p \in \Upsilon'_\ell$,
(a16) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}_{\mathbf{re}}} \; \bigcup_{i=1,i\neq\ell}^{N} \Upsilon_i \cup \Upsilon'_\ell$

By (a14), (a15), $\mathcal{E}_\gamma$, (a16), and $\mathcal{E}_\epsilon$, we have
(a17) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn^\partial_{ext}} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn^\partial_\ell} \cdots \mathcal{S}_{sn_N} \; \boldsymbol{\mathcal{R}_\mathcal{C}} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm^\partial_\ell} \cdots \mathcal{S}_{cm_N}$

By (a13) and (a17),
The conclusion follows

Subcase B: $u_{cm_\ell}.\mathsf{createFlag} = \mathit{NCreate}$**.**
By $\mathcal{E}_\beta$ we have
(a1) $\Gamma \vdash u_{sn_\ell} \sim_u u_{cm_\ell}$
By assumption,
(a2) the last rule the derived (a1) was U-IND
By the above and inversion we have:
(a3) $\Gamma \vdash tr_q : q(@\iota_q, \vec{t_q}) \sim_u \langle q(@\iota_q, \vec{t_q}), \mathsf{createFlag}, \mathtt{eID}, \lambda_q \rangle$

We define
(a4) $\forall i \in [1,n]$, $\mathtt{vID}_i \triangleq \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(a5) $ruleargs_p \triangleq rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(a6) $\mathtt{HrID}_p \triangleq \mathsf{hash}(ruleargs_p)$
(a7) $\flat_p \triangleq \mathsf{hash}(\lambda_q)$
(a8) $\lambda_p \triangleq \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$
(a9) $ruleExec_p \triangleq \langle \lambda_p, ruleargs_p, \lambda_q \rangle$
(a10) $u_{cm'_\ell} \triangleq \langle p(@\ell_p, \vec{x}_p)\sigma, \mathit{Create}, \mathtt{eID}, \lambda_p \rangle$.
(a11) If $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{cm^\partial_{in}} \triangleq [u_{cm'_\ell}]$ and $\mathcal{U}_{cm^\partial_{ext}} \triangleq [])$ else $(\mathcal{U}_{cm^\partial_{in}} \triangleq []$ and $\mathcal{U}_{cm^\partial_{ext}} \triangleq [u_{cm'_\ell}])$
(a12) $\Upsilon'_\ell = \Upsilon_\ell$

Using the above and CM-NCREATE we obtain
(a13) $singleCompressionCM(@\iota_\ell, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm^\partial_{in}}, \mathcal{U}_{cm^\partial_{ext}}, \Upsilon'_\ell)$.

If $yl_q \subseteq \bigcup_{i=1}^{N} \Upsilon_i$:
Case i: $ruleExec_p \in \Upsilon_\ell$
By $\mathcal{E}_\delta$
(a14) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}_\ell \; \boldsymbol{\mathcal{R}_{\mathbf{re}}} \; \bigcup_{i=1}^{N} \Upsilon_i$
Case ii: $ruleExec_p \notin \Upsilon_\ell$
By Each update that does not create rule provenances has a counterpart (Lemma 14)
$\exists u_{cm^\nu_\ell} \in \mathcal{U}_{cm_\ell}$ s.t. $u_{cm^\nu_\ell}.\mathsf{createFlag} = \mathit{Create}$
By definition of $ruleExec_p$
$DQ, \Gamma \vdash u_{cm^\nu_\ell} \nrightarrow ruleExec_p, u_{cm^\ell}[\mathsf{createFlag} \mapsto \mathit{Create}]$
By the above and $\mathcal{E}_\delta$
(a15) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \cup u_{cm^\nu_\ell} \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}_{\mathbf{re}}} \; \bigcup_{i=1}^{N} \Upsilon_i$

If $yl_q \nsubseteq \bigcup_{i=1}^{N} \Upsilon_i$:
In this case not all of yl_q has been fully derived yet.
Therefore there is some update already in the set of updates in the network that will eventually generate all of yl_q

By the Semi-naïve transition rules

$tr_q \in \mathcal{M}_\ell$
By the above and $\mathcal{E}_\delta$,
$\Gamma \vdash tr_q \sim_d yl_q :: ruleExec_p$
Thus given $\mathcal{E}_1$ and $\mathcal{E}_2$
$\exists u_{cm}{}^{\nu} \in \mathcal{U}_{cm}{}^{F}$, $\exists yl_A$, $\exists yl_B$ s.t.
$u_{cm}{}^{\nu}.\mathsf{createFlag} = Create$
and $yl_q = yl_A \circ yl_B$
and $DQ, \Gamma \vdash u_{cm}{}^{\nu} \looparrowright yl_B$
and $yl_A \subseteq \bigcup_{i=1}^{N} \Upsilon_i$
By definition of $ruleExec_p$,
$DQ, \Gamma \vdash u_{cm\ell} \Rightarrow ruleExec_p, u'_{cm\ell}[\mathsf{createFlag} \mapsto Create]$
By the above constructs
$DQ, \Gamma \vdash u_{cm}{}^{\nu} \looparrowright yl_B :: ruleExec_p$
Given that $\Gamma \vdash tr_p \sim_d yl_q :: ruleExec_p$ and the above and $\mathcal{E}_\delta$,
(a16) $\Gamma, DQ, \mathcal{U}_{cm}{}^{F} \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}_{re}} \; \bigcup_{i=1}^{N} \Upsilon_i$

By $\mathcal{E}_\alpha$, (a13), $\mathcal{E}_\gamma$, (a14)/(a15)/(a16), and $\mathcal{E}_\epsilon$, we have
(a17) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn}{}^{\partial}_{ext} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell}{}^{\partial} \cdots \mathcal{S}_{snN} \; \boldsymbol{\mathcal{R}_{\mathcal{C}}} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm}{}'_{ext} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell}{}^{\partial} \cdots \mathcal{S}_{cmN}$

By (a13) and (a17),
The conclusion follows

□

Lemma 14 (Each update that does not create rule provenances has a counterpart)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell} \cdots \mathcal{S}_{snN} \; \boldsymbol{\mathcal{R}_{\mathcal{C}}} \; \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN}$,
where $\mathcal{S}_{sn\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm\ell} = \langle @\iota_\ell, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $u_{cm\ell}.\mathsf{createFlag} = NCreate$
and $u_{sn\ell} = tr_p{:}P$
and $\Gamma \vdash tr_p{:}P \sim_d yl_p$
and $\mathsf{tail}(yl_p) \notin \bigcup_{i=1}^{N} \Upsilon_i$
implies
$\exists u_{cm\ell}^{\nu} \in \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} u_{cmi}$ *s.t.*
$DQ, \Gamma \vdash u_{cm\ell}^{\nu} \looparrowright yl_p^{\nu}$, *where* $yl_p = \_ \circ yl^{\nu}$

Proof. By inversion over the rule that last derived $u_{cm\ell}$

Case A: CM-Init-Event **was the last rule that derived** $u_{cm\ell}$**.**
By inversion on rule SN-Init-Event and since $\mathsf{createFlag} = NCreate$
(a1) $heq \in equiSet_\ell$, where $heq = \textsc{EquiHash}(ev, \Gamma)$
By (a1),
(a2) previously SN-Init-Event was fired to create some $u_{sn}{}^{\nu}$ s.t.
$u_{sn\ell}^{\nu} = u_{sn\ell}[\mathsf{createFlag} \mapsto NCreate]$
and $u_{sn\ell}^{\nu} \in \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} u_{cmi}$

Case B: CM-RuleFire-Intm **was the last rule that derived** $u_{cm\ell}$**.**
By the rule and given $\Gamma \vdash tr_p{:}P \sim_d yl_p$,
(b1) $yl_p = \_ :: ruleExec_q :: ruleExec_p$

If $ruleExec_q \in \bigcup_{i=1}^{N} \Upsilon_i$:
The last transition rule that derived $ruleExec_q$ also generated $u_{sn\ell}[\mathsf{createFlag} \mapsto Create]$
where $u_{sn\ell}[\mathsf{createFlag} \mapsto Create] \in \bigcup_{i=1}^{N} \Upsilon_i$
By the above,
$DQ \vdash u_{sn\ell}[\mathsf{createFlag} \mapsto Create] \looparrowright ruleExec_p$

If $ruleExec_q \notin \bigcup_{i=1}^{N} \Upsilon_i$:
By I.H. there is some $u_{sn}{}^{\nu}$
where $u_{sn}{}^{\nu}.\mathsf{createFlag} = Create$
and $DQ \vdash u_{sn}{}^{\nu} \looparrowright \_ :: ruleExec_q$
and $\_ :: ruleExec_q \subseteq yl_p$
By the above
$DQ \vdash u_{sn}{}^{\nu} \looparrowright \_ :: ruleExec_q :: ruleExec_p$ as required

Case C: CM-RuleFire-Interest **was the last rule that derived** $u_{cm\ell}$**.**

Because this rule never derives a new rule provenance no matter what the value of createFlag is, the lemma vacuously holds.

□

G.1.3 Online compression execution simulates semi-naïve evaluation

We show that online compression xecution simulates semi-naïve evaluation. To do so, for each set of transition rules for Online Compression execution, we state and prove a lemma that shows that these rules have a corresponding counterpart in Semi-Naïve evaluation. If initially the network configuration for both systems relate, after Online Compression execution steps to a new configuration, then Semi-Naïve evaluation is also able to step to a corresponding new configuration.

We present the necessary lemmas below, but omit most of the proof details as they are similar to those presented in Appendix G.1.2. Only the proof of *singleCompressionCM* simulates *singleDerivSN* (Lemma 21) is explained in detail as this is the key lemma that handles the updates of rule provenances. This lemma shows that given that the network configuration for both systems relate ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$), *singleCompressionCM*$(@_{\iota_q}, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell)$ takes in an update $u_{cm\ell}$ for tuple q, a rule r in the program DQ, a substitution σ for r, and returns a new update $u_{cm\ell}'$ and increments the set of rule provenances. As with *singleDerivSN* simulates *singleCompressionCM* (Lemma 13), the proof is rather complicated due to potential out of order executions. We explain the steps at a high level below. To prove this lemma, there are several cases to consider:

Case I: $u_{cm\ell}$ **represents a tuple that is an instance of the input event relation.**

Subcase A: $u_{cm\ell}$.createFlag = *Create***.**

By assumption, the last transition rule execute by the Online Compression execution was CM-Create. Given $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ and the above, we deduce the constructs for Semi-Naïve evaluation used to execute the corresponding transition rule SN-SingleSubst-Event. Because only one rule in DQ has been fired so far, it is easy to relate the new rule provenance and new update for both systems.

Subcase B: $u_{cm\ell}$.createFlag = *NCreate***.**

By assumption, the last transition rule execute by the Online Compression execution was CM-NCreate. Given $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$ and the above, we deduce the constructs for Semi-Naïve evaluation used to execute the corresponding transition rule SN-SingleSubst-Event. Because only one rule in DQ has been fired so far, it is easy to relate the new updates for both systems.

However, showing that the rule provenances relate is more involved provenance for Online Compression are not stored in this execution. There are two cases to consider. (1) There are no additions to the set of rule provenances for Online Compression execution as they have already been created and stored by past updates. In this case it is obvious that the set of rule provenances relate. (2) The rule provenance for Online Compression execution has not yet been created, by the corresponding provenance tree for Semi-Naïve evaluation is created and stored. By examining the rules for Online Compression execution, there is an enqueued update that will eventually create the required rule provenance.

Case II: $u_{cm\ell}$ **represents a tuple that is an instance of a fast-changing relation or a relation of interest.**

Subcase A: $u_{cm\ell}$.createFlag = *Create***.**

Similar argument to Case I, Subcase A, except that we additionally need to use the fact that $u_{cm\ell}$ relates to $u_{sn\ell}$, and that $u_{sn\ell}$ represents a provenance tree that is stored in the set of rule provenances in $\mathcal{C}_{sn}$ to show that the new update and rule provenance derived again relate.

Subcase B: $u_{cm\ell}$.createFlag = *NCreate***.**

Simmilar argument to Case I, Subcase B. Also uses the fact that $u_{cm\ell}$ relates to $u_{sn\ell}$, and that $u_{sn\ell}$ represents a provenance tree that is stored in the set of rule provenances in $\mathcal{C}_{sn}$.

Lemma 15 (Multi-step transition: online compression simulates semi-naïve)**.**
$\forall k \in \mathbb{N}$,
$\mathcal{C}_{init} \rightarrow^0_{SN} \mathcal{C}_{init} \rightarrow^1_{SN} \cdots \rightarrow^k_{SN} \mathcal{C}_{cm_{k+1}}$
implies
$\exists \mathcal{C}_{sn_{k+1}}$ *s.t.*
$\mathcal{C}_{init} \rightarrow^0_{SN} \mathcal{C}_{init} \rightarrow^1_{SN} \cdots \rightarrow^k_{SN} \mathcal{C}_{sn_{k+1}}$
and $\mathcal{C}_{cm_{k+1}}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{sn_{k+1}}$.

Proof. By induction over k and using Single-step transition: online compression simulates semi-naïve (Lemma 16). □

Lemma 16 (Single-step transition: online compression simulates semi-naïve)**.**
$\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$
and $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$
implies
$\exists \mathcal{C}_{sn}'$ *s.t.*
$\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$
and $\mathcal{C}_{sn}'\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}'$.

Proof. By inversion on rules for $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$ using Single-step transition per node: online compression simulates semi-naïve (Lemma 17), and applying the rules for $\mathcal{C}_{tcm} \nearrow_{CM} \mathcal{C}_{tcm}'$. □

Lemma 17 (Single-step transition per node: online compression simulates semi-naïve)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1}, \cdots, \mathcal{S}_{cm_\ell}, \cdots, \mathcal{S}_{cm_N}$
and $\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm_\ell}', \mathcal{U}_{cm_{ext}}'$
implies
$\exists \mathcal{U}_{sn_{ext}}', \exists \mathcal{S}_{sn_\ell}'$ *s.t.*
$\mathcal{S}_{sn_\ell} \hookrightarrow \mathcal{S}_{sn_\ell}', \mathcal{U}_{sn_{ext}}'$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}' \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}' \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}' \cdots \mathcal{S}_{cm_N}$.

Proof. By inversion on rules for $\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm_\ell}', \mathcal{U}_{cm_{ext}}'$, using *fireRulesCM* simulates *fireRulesSN* (Lemma 18), and applying the rules for $\mathcal{S}_{sn_\ell} \hookrightarrow \mathcal{S}_{sn_\ell}', \mathcal{U}_{sn_{ext}}'$. □

Lemma 18 (*fireRulesCM* simulates *fireRulesSN*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \mathcal{U}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \mathcal{U}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\bar{DQ} \subseteq DQ$
and $fireRulesCM(@\iota_q, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$
implies
$\exists \mathcal{U}_{sn_{in}}', \exists \mathcal{U}_{sn_{ext}}', \exists \mathcal{M}_\ell'$ *s.t.*
$fireRulesSN(@\iota_q, \Delta\bar{DQ}, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}_\ell')$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}' \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{sn_\ell} \circ \mathcal{U}_{sn_{in}}', equiSet_\ell, \mathcal{M}_\ell', \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell} \circ \mathcal{U}_{cm_{in}}', equiSet_\ell, \Upsilon_\ell', \Upsilon_{prov_\ell} \rangle$.

Proof. By induction over length of $\bar{DQ}$, inversion on the rules for $fireRulesCM(@\iota_q, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$, using *fireSingleRuleCM* simulates *fireSingleRuleSN* (Lemma 19), and applying the rules for *fireRulesSN*. □

Lemma 19 (*fireSingleRuleCM* simulates *fireSingleRuleSN*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \mathcal{U}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \mathcal{U}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $fireSingleRuleCM(@\iota_q, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$
implies
$\exists \mathcal{U}_{sn_{in}}', \exists \mathcal{U}_{sn_{ext}}', \exists \mathcal{M}_\ell'$ *s.t.*
$fireSingleRuleSN(@\iota_q, \Delta r, u_{sn_\ell}, \mathcal{DB}_\ell, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}_\ell')$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}' \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm}{}^{\partial} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{sn_\ell} \circ \mathcal{U}_{sn_{in}}', equiSet_\ell, \mathcal{M}_\ell', \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm_\ell} \circ \mathcal{U}_{cm_{in}}', equiSet_\ell, \Upsilon_\ell', \Upsilon_{prov_\ell} \rangle$.

Proof. By inversion on the rules for $fireSingleRuleCM(@\iota_q, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$, using *compressionCM* simulates *derivationSN* (Lemma 20), and applying the rules for *fireSingleRuleSN*. □

Lemma 20 (*compressionCM* simulates *derivationSN*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots, \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \mathcal{U}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \mathcal{U}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t_q}), \mathcal{DB}_\ell)$
and $\Sigma' \subseteq sel(\Sigma, \Delta r)$
and $compressionCM(@\iota_q, \Sigma', \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$
implies
$\exists \mathcal{U}_{sn_{in}}', \exists \mathcal{U}_{sn_{ext}}', \exists \mathcal{M}_\ell'$ *s.t.*
and $derivationSN(@\iota_q, \Sigma', \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn_{in}}', \mathcal{U}_{sn_{ext}}', \mathcal{M}_\ell')$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn_{ext}}' \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm_{ext}}' \rhd \mathcal{S}_{cm_1}, \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots, \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{sn_\ell} \circ \mathcal{U}_{sn_{in}}', equiSet_\ell, \mathcal{M}_\ell', \mathcal{M}_{prov_\ell} \rangle$
and $\mathcal{S}_{cm_\ell}^{\partial} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm_\ell} \circ \mathcal{U}_{cm_{in}}', equiSet_\ell, \Upsilon_\ell', \Upsilon_{prov_\ell} \rangle$.

Proof. By induction on the length of Σ', inversion on the rules for $compressionCM(@\iota_q, \Sigma', \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm_{in}}', \mathcal{U}_{cm_{ext}}', \Upsilon_\ell')$, using *singleCompressionCM* simulates *singleDerivSN* (Lemma 21), and applying the rules for *derivationSN*. □

Lemma 21 (*singleCompressionCM* simulates *singleDerivSN*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell}\rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell}\rangle$
and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
and $\Sigma' \in sel(\Sigma, \Delta r)$
and $\sigma \in \Sigma'$
and $singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
implies
$\exists \mathcal{U}_{sn'_{in}}, \exists \mathcal{U}_{sn'_{ext}}, \exists \mathcal{M}'_\ell$ *s.t.*
$singleDerivSN(@\iota_q, \sigma, \Delta r, u_{sn_\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$
and $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}^{\partial}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}^{\partial}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}) \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \mathcal{M}'_\ell, \mathcal{M}_{prov_\ell}\rangle$
and $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}) \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell}\rangle$.

Proof.
Assume that
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$
where $\mathcal{S}_{sn_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{sn_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \mathcal{M}_\ell, \mathcal{M}_{prov_\ell}\rangle$
and $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell}\rangle$
(2) $r \in DQ$
(3) $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
(4) $\Sigma' \in sel(\Sigma, \Delta r)$
(5) $\sigma \in \Sigma'$
(6) $singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$

By inversion on the rules for $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N}$ we have
$\forall i \in [1, N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i}\rangle$
$\forall i \in [1, N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i}\rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_U}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N], \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_U}\ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{re}}\ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{prov}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$.

By inversion on the rules for $\mathcal{E}_\delta$, we have
$\mathcal{E}_1 :: \Gamma, DQ \vdash \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i} \frown \Upsilon^F$
$\mathcal{E}_2 :: \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{M}_i \approx_d \bigcup_{i=1}^{N} \Upsilon_i \cup \Upsilon^F$.

By inversion on the rules for (6), there exists constructs s.t.
$u_{cm_\ell} = \langle q(@\iota_q, \vec{t}_q), \mathsf{createFlag}, \mathtt{eID}, \lambda_q\rangle$

Case I: $\Gamma(q)[tuple] = \mathsf{event}$**.**

Case A: $u_{cm_\ell}.\mathsf{createFlag} = Create$**.**
By assumption
The last rule that derived (6) was CM-Create

By inversion we have
(a1) $\Delta r = rID\ p(@\ell_p, \vec{x}_p) \text{ :- } q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(a2) $u_{cm_\ell} = \langle q(@\iota_q, \vec{t}_q), Create, \mathtt{eID}, \lambda_q\rangle$
(a3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q)$
(a4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(a5) $\forall i \in [1, n], \mathtt{vID}_i = \text{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(a6) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(a7) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(a8) $heq = \text{EquiHash}(q(@\ell_q, \vec{x}_q), \Gamma)$
(a9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$
(a10) $u_{cm'_\ell} = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p\rangle$
(a11) $ruleExec_p = \langle \lambda_p, ruleargs, \lambda_q\rangle$
(a12) $\Upsilon'_\ell = \Upsilon_\ell \cup \{ruleExec_p\}$
(a13) if $\sigma(@\ell_p) = @\iota_q$ then $\mathcal{U}_{cm'_{in}} = [u_{cm'_\ell}], \mathcal{U}_{cm'_{ext}} = []$ else $\mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm'_\ell}]$

We use the above constructs to define:
(a14) $u_{sn\ell} \triangleq q(@\iota_q, \vec{t_q})$
(a15) $tr_p \triangleq (rID, p(@\ell_p, \vec{x}_p)\sigma, q(@\iota_q, \vec{t_q}), b_1(@\ell_q, \vec{x}_{b1})\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$
(a16) $u_{sn'_\ell} \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$
(a17) if $(\sigma(@\ell_p) = @\iota_q)$ then $\mathcal{U}_{sn'_{in}} \triangleq [u_{sn'_\ell}], \mathcal{U}_{cm'_{ext}} \triangleq []$ else $\mathcal{U}_{sn'_{in}} \triangleq [], \mathcal{U}_{sn'_{ext}} \triangleq [u_{sn'_\ell}]$
(a18) $\mathcal{M}'_\ell \triangleq \mathcal{M}_\ell \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

Using the above constructs we apply SN-SingleSubst to obtain:
(a19) $singleDerivSN(@\iota_\ell, \sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$

By definition of the constructs,
$\Gamma \vdash u_{sn'_\ell} \sim_u u_{cm'_\ell}$
By $\mathcal{E}_\alpha$ and the above,
(a20) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}}\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}}$
By $\mathcal{E}_\beta$ and the above,
(a21) $\forall i \in [1, N]$, $\Gamma \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{sn_i} \cup ((u_{sn\ell} :: \bar{\mathcal{U}_{sn\ell}}) \circ \mathcal{U}_{sn'_\ell})\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup ((u_{cm\ell} :: \bar{\mathcal{U}_{cm\ell}}) \circ \mathcal{U}_{cm'_\ell})$

By definition of tr_p,
$\Gamma \vdash tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \sim_d ruleExec_p$
By $\mathcal{E}_\delta$ the above,
(a22) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \Upsilon'_\ell \cup \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{re}}\ \mathcal{M}'_\ell \cup \bigcup_{i=1, i\neq\ell}^{N} \Upsilon_i$

By (a20), (a21), $\mathcal{E}_\gamma$, (a23), $\mathcal{E}_\epsilon$,
(a24) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_\mathcal{C}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cm_N}$

By (a19) and (a24),
The conclusion holds

Case B: $u_{cm\ell}.\mathsf{createFlag} = NCreate$**.**
By assumption
the last rule that derived (6) was CM-NCreate.
By inversion on that rule,
(b1) $\Delta r = rID\ p(@\ell_p, \vec{x}_p) \text{ :- } q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(b2) $u_{cm\ell} = \langle q(@\iota_q, \vec{t_q}), Create, \mathtt{eID}, \lambda_q \rangle$
(b3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t_q})$
(b4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(b5) $\forall i \in [1, n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(b6) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(b7) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(b8) $heq = \textsc{EquiHash}(q(@\ell_q, \vec{x}_q), \Gamma)$
(b9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, heq)$
(b10) $u_{cm'_\ell} = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p \rangle$
(b11) $\Upsilon'_\ell = \Upsilon_\ell$
(b12) if $(\sigma(@\ell_p) == @\iota_q)$ then $\mathcal{U}_{cm'_{in}} = [u_{cm'_\ell}], \mathcal{U}_{cm'_{ext}} = []$ else $\mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm'_\ell}]$

We use the above to define the following constructs for Semi-Naive Evaluation
(b13) $u_{sn\ell} \triangleq q(@\iota_q, \vec{t_q})$
(b14) $tr_p \triangleq (rID, p(@\ell_p, \vec{x}_p)\sigma, q(@\iota_q, \vec{t_q}), b_1(@\ell_q, \vec{x}_{b1})\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$
(b15) $u_{sn'_\ell} \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$
(b16) if $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{sn'_{in}} \triangleq [u_{sn'_\ell}]$ and $\mathcal{U}_{sn'_{ext}} \triangleq [])$ else $(\mathcal{U}_{sn'_{in}} \triangleq []$ and $\mathcal{U}_{sn'_{ext}} \triangleq [u_{sn'_\ell}])$
(b17) $\mathcal{M}'_\ell \triangleq \mathcal{M}_\ell \cup tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

We apply SN-SingleSubst to obtain
(b18) $singleDerivSN(@\iota_\ell, \sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$

By our definitions
$\Gamma \vdash u_{sn'_\ell} \sim_u u_{cm'_\ell}$
By $\mathcal{E}_\alpha$ and the above,
(b19) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}}\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}}$
By $\mathcal{E}_\beta$ and the above,
(b20) $\forall i \in [1, N]$, $\Gamma \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{sn_i} \cup ((u_{sn\ell} :: \bar{\mathcal{U}_{sn\ell}}) \circ \mathcal{U}_{sn'_{in}})\ \boldsymbol{\mathcal{R}_\mathcal{U}}\ \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup ((u_{cm\ell} :: \bar{\mathcal{U}_{cm\ell}}) \circ \mathcal{U}_{cm'_{in}})$

Additionally we define the following constructs for Online Compression Evaluation
(b19) $heq = \textsc{EquiHash}(q(@\iota_q, \vec{t_q}), \Gamma)$
(b20) $\lambda_q = \mathsf{id}(\emptyset, \emptyset, heq)$
(b21) $ruleExec_p \triangleq \langle \lambda_p, ruleargs_p, \lambda_q \rangle$

If $ruleExec_p \in \Upsilon_\ell$:
By $\mathcal{E}_\delta$ we have
(b22) $\Gamma, DQ, {\mathcal{U}_{cm}}^F \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}_{re}} \; \bigcup_{i=1}^{N} \Upsilon_i$

If $ruleExec_p \notin \Upsilon_\ell$:
By examining the rules,
rule CM-Init-Event was fired in the past
and $\exists u{cm}_\ell^\mu \in \mathcal{U}_{cm\ell}$ s.t. $u{cm}_\ell^\mu = u_{cm\ell}[\mathsf{createFlag} \mapsto Create]$
By construction,
$\Gamma \vdash u{cm}_\ell^\mu \looparrowright ruleExec_p, u{cm}'_\ell[\mathsf{createFlag} \mapsto Create]$
By the above and $\mathcal{E}_\delta$ thus
(b23) $\Gamma, DQ, {\mathcal{U}_{cm}}^F \cup u{cm}_\ell^\mu \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}_{re}} \; \bigcup_{i=1}^{N} \Upsilon_i$

By (b19), (b20) $\mathcal{E}_\gamma$, (b22)/(b23) and $\mathcal{E}_\epsilon$, we have
(b26) $\mathcal{Q}_{sn} \circ {\mathcal{U}_{sn}}_{ext}^{\partial} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^{\partial} \cdots \mathcal{S}_{sn_N} \; \boldsymbol{\mathcal{R}_C} \; \mathcal{Q}_{cm} \circ {\mathcal{U}_{cm}}'_{ext} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^{\partial} \cdots \mathcal{S}_{cm_N}$

By (b18) and (b26),
the conclusion holds

Case II: $\Gamma(q)[tuple] = \mathsf{fast}$**.**

Case A: $u_{cm\ell}.\mathsf{createFlag} = Create$**.**
By assumption
the last rule that derived (6) was CM-Create
By inversion on that rule
(a1) $\Delta r = rID\ p(@\ell_p, \vec{x}_p) \coloneq q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(a2) $u_{cm\ell} = \langle q(@\iota_q, \vec{t_q}), Create, \mathtt{eID}, \lambda_q \rangle$
(a3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t_q})$
(a4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(a5) $\forall i \in [1, n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(a6) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(a7) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(a8) $\flat_p = \mathsf{hash}(\lambda_q)$
(a9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$
(a10) $u{cm}'_\ell = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p \rangle$
(a11) $ruleExec_p = \langle \lambda_p, ruleargs, \lambda_q \rangle$
(a12) $\Upsilon'_\ell = \Upsilon_\ell \cup ruleExec_p$
(a13) if $(\sigma(@\ell_p) = @\iota_q)$ then $({\mathcal{U}_{cm}}'_{in} = [u{cm}'_\ell]$ and ${\mathcal{U}_{cm}}'_{ext} = [])$ else $({\mathcal{U}_{cm}}'_{in} = []$ and ${\mathcal{U}_{cm}}'_{ext} = [u{cm}'_\ell])$

By $\mathcal{E}_\beta$ and since $u_{cm\ell} \in \mathcal{U}_{cm\ell}$,
(a14) $\exists tr_q$ s.t. $u_{cm\ell} = tr_q$

Using the above we define
(a15) $tr_p \triangleq (rID, p(@\ell_p, \vec{x}_p)\sigma, tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\ell_q, \vec{x}_{b1})\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$
(a16) $u{sn}'_\ell \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$
(a17) if $(\sigma(@\ell_p) = @\iota_q)$ then $({\mathcal{U}_{sn}}'_{in} \triangleq [u{sn}'_\ell]$ and ${\mathcal{U}_{sn}}'_{ext} \triangleq [])$ else $({\mathcal{U}_{sn}}'_{in} \triangleq []$ and ${\mathcal{U}_{sn}}'_{ext} \triangleq [u{sn}'_\ell])$
(a18) $\mathcal{M}'_\ell \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

Using the above constructs we apply SN-SingleSubst to obtain:
(a19) $singleDerivSN(@\iota_\ell, \sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell) = ({\mathcal{U}_{sn}}'_{in}, {\mathcal{U}_{sn}}'_{ext}, \mathcal{M}'_\ell)$

By our definitions
$\Gamma \vdash u{sn}'_\ell \sim_u u{cm}'_\ell$
By $\mathcal{E}_\alpha$ and the above,
(a20) $\Gamma \vdash \mathcal{Q}_{sn} \circ {\mathcal{U}_{sn}}'_{ext} \; \boldsymbol{\mathcal{R}_\mathcal{U}} \; \mathcal{Q}_{cm} \circ {\mathcal{U}_{cm}}'_{ext}$
By $\mathcal{E}_\beta$ and the above,
(a21) $\forall i \in [1, N]$, $\Gamma \vdash \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{sni} \cup ((u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ {\mathcal{U}_{sn}}'_{in}) \; \boldsymbol{\mathcal{R}_\mathcal{U}} \; \bigcup_{i=1, i\neq\ell}^{N} \mathcal{U}_{cmi} \cup ((u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ {\mathcal{U}_{cm}}'_{in})$

By examining the rules for Semi-Naïve Evaluation and given $u_{sn\ell} = tr_q$,
(a22) $tr_q \in \mathcal{M}_\ell$
By the above, given $\mathcal{E}_\delta$ and since $\mathsf{createFlag} = \mathit{Create}$,
$\exists yl_q$ s.t.
$yl_q \subseteq \bigcup_{i=1}^{N} \Upsilon_i$
and $\Gamma \vdash tr_q \sim_d yl_q$
By the above and using the definitions of tr_p and $\mathit{ruleExec}_p$,
$\Gamma \vdash tr_p{:}p(@\ell_p, \vec{x}_p)\sigma \sim_d yl_q :: \mathit{ruleExec}_p$
By the above and $\mathcal{E}_\delta$,
(a23) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}_\ell \; \boldsymbol{\mathcal{R}_{re}} \; \bigcup_{i=1,i\neq\ell}^{N} \Upsilon_i \cup \Upsilon_\ell$

By (a20), (a21), $\mathcal{E}_\gamma$, (a23), $\mathcal{E}_\epsilon$,
(a24) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{sn\ell}^{\partial} \cdots \mathcal{S}_{snN} \; \boldsymbol{\mathcal{R}_C} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell}^{\partial} \cdots \mathcal{S}_{cmN}$

By (a20) and (a24),
the conclusion follows

Case B: $u_{cm\ell}.\mathsf{createFlag} = \mathit{NCreate}$**.**
By assumption
the last rule that derived (6) was CM-NCreate
By inversion on that rule
(b1) $\Delta r = \mathit{rID}\ p(@\ell_p, \vec{x}_p) \text{ :- } q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(b2) $u_{cm\ell} = \langle q(@\iota_q, \vec{t_q}), \mathit{Create}, \mathtt{eID}, \lambda_q \rangle$
(b3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t_q})$
(b4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(b5) $\forall i \in [1,n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(b6) $\mathit{ruleargs}_p = \mathit{rID} :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(b7) $\mathtt{HrID}_p = \mathsf{hash}(\mathit{ruleargs}_p)$
(b8) $\flat_p = \mathsf{hash}(\lambda_q)$
(b9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$
(b10) $u_{cm'_\ell} = \langle p(@\ell_p, \vec{x}_p)\sigma, \mathit{Create}, \mathtt{eID}, \lambda_p \rangle$
(b11) $\mathit{ruleExec}_p = \langle \lambda_p, \mathit{ruleargs}, \lambda_q \rangle$
(b12) $\Upsilon'_\ell = \Upsilon_\ell$
(b13) if $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{cm'_{in}} = [u_{cm'_\ell}]$ and $\mathcal{U}_{cm'_{ext}} = [])$ else $(\mathcal{U}_{cm'_{in}} = []$ and $\mathcal{U}_{cm'_{ext}} = [u_{cm'_\ell}])$

By $\mathcal{E}_\beta$ and since $u_{cm\ell} \in \mathcal{U}_{cm\ell}$,
(b14) $\exists tr_q$ s.t. $u_{cm\ell} = tr_q$

Using the above constructs we define
(b15) $tr_p \triangleq (\mathit{rID}, p(@\ell_p, \vec{x}_p)\sigma, tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\ell_q, \vec{x}_{b1})\sigma :: \cdots :: b_n(@\ell_q, \vec{x}_{bn})\sigma)$
(b16) $u_{sn'_\ell} \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$
(b17) if $(\sigma(@\ell_p) = @\iota_q)$ then $(\mathcal{U}_{sn'_{in}} \triangleq [u_{sn'_\ell}]$ and $\mathcal{U}_{sn'_{ext}} \triangleq [])$ else $(\mathcal{U}_{sn'_{in}} \triangleq []$ and $\mathcal{U}_{sn'_{ext}} \triangleq [u_{sn'_\ell}])$
(b18) $\mathcal{M}'_\ell \triangleq tr_p{:}p(@\ell_p, \vec{x}_p)\sigma$

Using the above we apply SN-SingleSubst to obtain:
(b19) $\mathit{singleDerivSN}(@\iota_\ell, \sigma, \Delta r, u_{sn\ell}, \mathcal{M}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{M}'_\ell)$

By our definitions
$\Gamma \vdash u_{sn'_\ell} \sim_u u_{cm'_\ell}$
By $\mathcal{E}_\alpha$ and the above,
(b20) $\Gamma \vdash \mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \; \boldsymbol{\mathcal{R}_U} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}}$
By $\mathcal{E}_\beta$ and the above,
(b21) $\forall i \in [1,N]$, $\Gamma \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{U}_{sn_i} \cup ((u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ \mathcal{U}_{sn'_{in}}) \; \boldsymbol{\mathcal{R}_U} \; \bigcup_{i=1,i\neq\ell}^{N} \mathcal{U}_{cm_i} \cup ((u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ \mathcal{U}_{cm'_{in}})$

By examining the rules for Semi-Naïve Evaluation and given $u_{sn\ell} = tr_q$,
(b22) $tr_q \in \mathcal{M}_\ell$

If $yl_q \subseteq \bigcup_{i=1}^{N} \Upsilon_i$:
Case i: $\mathit{ruleExec}_p \in \Upsilon_\ell$
By $\mathcal{E}_\delta$
(b23) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1,i\neq\ell}^{N} \mathcal{M}_i \cup \mathcal{M}_\ell \; \boldsymbol{\mathcal{R}_{re}} \; \bigcup_{i=1}^{N} \Upsilon_i$

Case ii: $ruleExec_p \notin \Upsilon_\ell$

By examining the rules for Online Compression,

$\exists ucm_\ell^\nu \in \mathcal{U}_{cm_\ell}$ s.t. $ucm_\ell^\nu.\mathsf{createFlag} = Create$

By definition of $ruleExec_p$

$DQ, \Gamma \vdash ucm_\ell^\nu \not\rightarrow ruleExec_p, ucm^\ell[\mathsf{createFlag} \mapsto Create]$

By the above and $\mathcal{E}_\delta$

(b24) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \cup ucm_\ell^\nu \vdash \bigcup_{i=1, i\neq\ell}^N \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}}_{\mathbf{re}} \; \bigcup_{i=1}^N \Upsilon_i$

If $yl_q :\not\subseteq \bigcup_{i=1}^N \Upsilon_i$:

By (b22), $\mathcal{E}_1$ and $\mathcal{E}_2$

$\exists ucm^\nu \in \mathcal{U}_{cm}{}^F$, $\exists yl_A$, $\exists yl_B$ s.t.

$ucm^\nu.\mathsf{createFlag} = Create$

and $yl_q = yl_A \circ yl_B$

and $DQ, \Gamma \vdash ucm^\nu \not\rightarrow yl_B$

and $yl_A \subseteq \bigcup_{i=1}^N \Upsilon_i$

By definition of $ruleExec_p$,

$DQ, \Gamma \vdash ucm_\ell \Rightarrow ruleExec_p, ucm'_\ell[\mathsf{createFlag} \mapsto Create]$

By the above constructs

$DQ, \Gamma \vdash ucm^\nu \not\rightarrow yl_B :: ruleExec_p$

Given that $\Gamma \vdash tr_p \sim_d yl_q :: ruleExec_p$ and the above and $\mathcal{E}_\delta$,

(b25) $\Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1, i\neq\ell}^N \mathcal{M}_i \cup \mathcal{M}'_\ell \; \boldsymbol{\mathcal{R}}_{\mathbf{re}} \; \bigcup_{i=1}^N \Upsilon_i$

By (b20), (b21), $\mathcal{E}_\gamma$, (b23)/(b24)/(b25), $\mathcal{E}_\epsilon$,

(b26) $\mathcal{Q}_{sn} \circ \mathcal{U}_{sn'_{ext}} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_\ell}^\partial \cdots \mathcal{S}_{sn_N} \; \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}} \; \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell}^\partial \cdots \mathcal{S}_{cm_N}$

By (b19) and (b26),

the conclusion follows

□

G.2 Bisimulation between the two online compression executions

Our overall goal is to show that there is a bisimulation relation between semi-naïve evaluation and the online compression execution that shares storage *across* equivalence classes. In this section, we show that there is a bisimulation relation between the online compression execution that shares storage *within* equivalence classes and the online compression execution that shares storage *across* equivalence classes. Together with the results from Appendix G.1, we reach our desired conclusion.

The main difference between the two versions of online compression is that online compression that shares storage *across* equivalence classes uses even less storage space to record rule provenances than online compression that shares storage *within* equivalence classes. It accomplishes this by storing the parent-child relation ship separately from the constructs used to execute a rule. Therefore the constructs used to execute a rule could potentially be shared across multiple equivalence classes. In contrast, online compression that shares storage *within* equivalence classes cannot share any rule provenance storage between different equivalence classes.

We prove the bisimulation between the two versions of online compression by formally defining define a relation $\sim\!\sim_{\mathcal{C}}$ between the network configuration $\mathcal{C}_{cm}$ of the online compression execution that shares storage *within* equivalence classes and the network configuration $\mathcal{C}_{tcm}$ of the online compression execution that shares storage *across* equivalence classes. Then, we show that $\mathcal{C}_{cm} \sim\!\sim_{\mathcal{C}} \mathcal{C}_{tcm}$ defines a bisimulation between the two executions.

G.2.1 Relating network states

Most constructs for online compression that shares storage *within* equivalence classes and online compression that shares storage *across* equivalence classes are identical. The constructs that handle rule provenance are necessarily different as the version that shares storage across equivalence classes optimizes storage even more. We explain how we relate the differing constructs below, using the Packet Forwarding example in Figure 28 to illustrate.

$$\begin{array}{lllll} r1 & \mathsf{packet}(@N, S, D, DT) & :- & \mathsf{packet}(@L, S, D, DT), \mathsf{route}(@L, D, N). \\ r2 & \mathsf{recv}(@L, S, D, DT) & :- & \mathsf{packet}(@L, S, D, DT), D == L. \end{array}$$

Figure 28: Packet Forwarding

In this example, we assume that the initial network configuration for both versions of online compression each have two slow changing tuples, $\mathsf{route}(@1, 3, 2)$ and $@2,3,3$. Assuming an input event tuple $\mathsf{packet}(@1, 1, 3, \mathrm{hi})$ triggers program execution, Figure 29 shows the rule provenances that are stored after online compression of the packet forwarding program that shares storage *within* equivalence classes terminates. The rule provenances generated are on the left column.

$heq = \text{EQUIHASH}(\mathsf{packet}(@1,1,3,\text{hi}))$	
$ruleExec_1 = \langle \lambda_1, ruleargs_1, \lambda_0 \rangle$	$ruleargs_1 = r1 :: 1 :: \text{TUPLEHASH}(\mathsf{route}(@1,3,2))$ $\mathtt{HrID}_1 = \mathsf{hash}(ruleargs_1)$ $\lambda_0 = \mathsf{id}(\emptyset, \emptyset, heq)$ $\lambda_1 = \mathsf{id}(@1, \mathtt{HrID}_1, heq)$
$ruleExec_2 = \langle \lambda_2, ruleargs_2, \lambda_1 \rangle$	$ruleargs_2 = r1 :: 2 :: \text{TUPLEHASH}(\mathsf{route}(@2,3,3))$ $\mathtt{HrID}_2 = \mathsf{hash}(ruleargs_2)$ $\lambda_2 = \mathsf{id}(@2, \mathtt{HrID}_2, \lambda_1)$
$ruleExec_3 = \langle \lambda_3, ruleargs_3, \lambda_2 \rangle$	$ruleargs_3 = r2 :: 3$ $\mathtt{HrID}_3 = \mathsf{hash}(ruleargs_3)$ $\lambda_3 = \mathsf{id}(@3, \mathtt{HrID}_3, \lambda_2)$

Figure 29: Rule provenance storage after online compression of the packet forwarding program that shares storage *within* equivalence classes terminates. The input event tuple is $\mathsf{packet}(@1,1,3,\mathbf{hi})$.

Figure 30 shows the rule provenances that are stored after online compression of the packet forwarding program that shares storage *across* equivalence classes terminates. The rule provenances generated are in the first two columns. The corresponding rule provenance for online compression that shares storage *across* equivalence classes is in the right column.

Sharing *across* equivalence classes		Sharing *within* equivalence classe
Provenance of an individual rule	Parent-child relation	Provenance of individual rule and parent-child relation combined
$(\langle \lambda_1{:}1, \lambda_1{:}2 \rangle, ruleargs_1)$	(λ_1, λ_0)	$ruleExec_1$
$(\langle \lambda_2{:}1, \lambda_2{:}2 \rangle, ruleargs_2)$	(λ_2, λ_1)	$ruleExec_2$
$(\langle \lambda_3{:}1, \lambda_3{:}2 \rangle, ruleargs_3)$	(λ_3, λ_2)	$ruleExec_3$

Figure 30: Rule provenance storage after online compression of the packet forwarding program that shares storage *across* equivalence classes terminates. The input event tuple is $\mathsf{packet}(@1,1,3,\mathbf{hi})$.

Relating a rule provenance element ($ruleExec \sim\!\sim_\ell lcm :: ncm$).

Online compression that shares storage *within* equivalence classes records the arguments used to fire a DELP rule and the parent-child relationship between the rule provenance representing the previous rule fired together as *ruleExec*. *ruleExec* has form $\langle \lambda_p, ruleargs_p, \lambda_q \rangle$, in which λ_p (where $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$) is a unique identifier for *ruleExec*, $ruleargs_p$ contains the necessary constructs to fire a rule, and λ_q stores the unique identifier for the previous rule fired.

In contrast, online compression that shares storage *across* equivalence classes records those two pieces of information separately in order to further compress the provenances. The arguments used to fire a DELP rule are recorded as *ncm*, which may be used to record the provenance of executions belonging to multiple equivalence classes. *lcm* is used solely to record the parent-child relationship between the rule provenances, and cannot be shared between multiple equivalence classes. Figures 29 and 30 provide a concrete example of how to relate an *ruleExec* element to a node element *ncm* and link element *lcm*.

Relating sets of rule provenances ($\Upsilon \sim\!\sim_{ruleExec} \mathcal{L}; \mathcal{N}$).

The base case is when no provenances have been recorded and Υ, $\mathcal{L}$, and $\mathcal{N}$ are empty sets.

In the inductive case, every rule provenance *ruleExec* in Υ relates to a node provenance *ncm* in $\mathcal{N}$ and parent-child provenance *lcm* in $\mathcal{L}$.

For example, if $\Upsilon = \{ruleExec_1, ruleExec_2\}$ and $\mathcal{L} = \{(\lambda_1, \lambda_0), (\lambda_2, \lambda_1); \mathcal{N} = \{(\langle \lambda_1{:}1, \lambda_1{:}2 \rangle, ruleargs_1), (\langle \lambda_2{:}1, \lambda_2{:}2 \rangle, ruleargs_2)\}$ and then $\Upsilon \sim\!\sim_{ruleExec} \mathcal{L}; \mathcal{N}$, and given $ruleExec_3 \sim\!\sim_\ell (\lambda_3, \lambda_2) :: (\langle \lambda_3{:}1, \lambda_3{:}2 \rangle, ruleargs_3)$, then $\Upsilon \cup ruleExec_3 \sim\!\sim_{ruleExec} \mathcal{L} \cup (\lambda_3, \lambda_2); \mathcal{N} \cup (\langle \lambda_3{:}1, \lambda_3{:}2 \rangle, ruleargs_3)$.

Relating individual network states ($\mathcal{S}_{cm} \sim\!\sim_{\mathcal{S}} \mathcal{T}_{cm}$). Given a state $\mathcal{S}_{cm}$ for online compression that shares storage *within* equivalence classes and a state $\mathcal{T}_{cm}$ for online compression that shares storage *across* equivalence classes, if the constructs that store rule provenances for both states relate ($\Upsilon \sim\!\sim_{ruleExec} \mathcal{L}; \mathcal{N}$) and the other constructs in their states are identical, then these two states relate

Relating network configurations ($\mathcal{C}_{cm} \sim\!\sim_{\mathcal{S}} \mathcal{C}_{tcm}$). Given that all states in the two network configurations relate and the sets external updates for both network configurations are identical, then the network configurations relate.

$\boxed{ruleExec \sim\!\sim_\ell lcm :: ncm}$

$$\frac{\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)}{\langle \lambda_p, ruleargs_p, \lambda_q \rangle \sim\!\sim_\ell (\lambda_p, \lambda_q) :: (\langle @\iota_q, \mathtt{HrID}_p \rangle, ruleargs_p)} \sim\!\sim_\ell$$

$\boxed{\Upsilon \sim\sim_{ruleExec} \mathcal{L};\mathcal{N}}$

$$\frac{}{\{\} \sim\sim_{ruleExec} \{\};\{\}}\ \sim\sim_{ruleExec}\text{-Base} \qquad \frac{ruleExec \sim\sim_{\ell} (lcm :: ncm) \qquad \Upsilon \sim\sim_{ruleExec} (\mathcal{L};\mathcal{N})}{\Upsilon \cup ruleExec \sim\sim_{ruleExec} (\mathcal{L} \cup lcm; \mathcal{N} \cup ncm)}\ \sim\sim_{ruleExec}\text{-Ind}$$

$\boxed{\mathcal{S}cm \sim\sim_{\mathcal{S}} \mathcal{T}cm}$

$$\frac{\Upsilon \sim\sim_{ruleExec} \mathcal{L};\mathcal{N}}{\langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}cm, equiSet, \Upsilon, \Upsilon_{prov}\rangle \sim\sim_{\mathcal{S}} \langle @\iota, DQ, \Gamma, \mathcal{DB}, \mathcal{E}, \mathcal{U}cm, equiSet, \mathcal{L}, \mathcal{N}, \Upsilon_{prov}\rangle}\ \sim\sim_{\mathcal{S}}$$

$\boxed{\mathcal{C}cm \sim\sim_{\mathcal{C}} \mathcal{C}tcm}$

$$\frac{\forall i \in [1, N], \mathcal{S}cm_i \sim\sim_{\mathcal{S}} \mathcal{T}cm_i}{\mathcal{Q}cm \rhd \mathcal{S}cm_1 \cdots \mathcal{S}cm_N \sim\sim_{\mathcal{C}} \mathcal{Q}cm \rhd \mathcal{T}cm_1 \cdots \mathcal{T}cm_N}\ \sim\sim_{\mathcal{C}}$$

G.2.2 *Relating provenance trees to ordered lists of provenances*

Next we define relations between provenance trees and ordered lists of rule provenances. We will use these relations to show that every time online compresion that shares storage *within* equivalent classes takes a step, online compresion that shares storage *across* equivalent classes takes a step and the bisimulation relation between the network state again holds, and vice versa.

Relating an ordered list of rule provenances ($yl \sim\sim_{ch} ch$).

The base case is when the ordered lists of rule provenances are empty.

In the inductive case, every *ruleExec* rule provenance in yl relates to the corresponding pair of $lcm :: ncm$ in ch.

Relating a provenance tree to an ordered list of rule provenances ($\Gamma \vdash tr \sim\sim_{d} ch$).

Given a provenance tree tr, if tr relates to an ordered list of rule provenances yl that store the parent-child relationships together with arguments to rules ($\Gamma \vdash tr \sim\sim_{d} yl$), and if yl relates to ch ($yl \sim\sim_{ch} ch$), then tr relates to ch ($\Gamma \vdash tr \sim\sim_{d} ch$).

$\boxed{yl \sim\sim_{ch} ch}$

$$\frac{}{[] \sim\sim_{ch} []}\ \sim\sim_{ch}\text{-Base} \qquad \frac{ruleExec \sim\sim_{\ell} (lcm :: ncm)}{yl :: ruleExec \sim\sim_{ch} ch \rightsquigarrow (lcm :: ncm)}\ \sim\sim_{ch}\text{-Ind}$$

$\boxed{\Gamma \vdash tr \sim\sim_{d} ch}$

$$\frac{\Gamma \vdash tr \sim\sim_{d} yl \qquad yl \sim\sim_{ch} ch}{\Gamma \vdash tr \sim\sim_{d} ch}\ \sim\sim_{d}$$

G.2.3 *Online compression sharing storage* within *equivalence classes simulates online compression sharing storage* across *equivalence classes*

We show that online compression that shares storage *within* equivalence classes simulates online compression that shares storage *across* equivalence classes. We show that given any network configuration $\mathcal{C}cm$ (where $\mathcal{C}cm = \mathcal{Q}cm \rhd \mathcal{S}cm_1 \cdots \mathcal{S}cm_i \cdots \mathcal{S}cm_N$) for Online Compression (via sharing storage within equivalence classes), there exists a corresponding network configuration $\mathcal{C}tcm$ (where $\mathcal{C}tcm = \mathcal{Q}cm \rhd \mathcal{T}cm_1 \cdots \mathcal{T}cm_i \cdots \mathcal{T}cm_N$) for Online Compression (via sharing storage across equivalence classes), such that $\mathcal{C}cm \sim\sim_{\mathcal{C}} \mathcal{C}tcm$.

To prove this, for each set of transition rules for online compression that shares storage *within* equivalence classes, we state and prove a lemma that shows that these rules have a corresponding counterpart in online compression that shares storage *across* equivalence classes. If initially the network configuration for both systems relate, after online compression that shares storage *within* equivalence classes steps to a new configuration, then online compression that shares storage *across* equivalence classes is also able to step to a corresponding new configuration.

We present the necessary lemmas below, but omit most of the proof details as they are similar to those presented in Appendix G.1.2. Only the proof of *singleCompressionCM* simulates *singleCompressionAcrossCM*] (Lemma 28) differs somewhat lemma that handles the updates of rule provenances. The proof exploits the fact that for every rule provenance element in $\mathcal{C}cm$, there is one corresponding rule provenance link and node in $\mathcal{C}tcm$ and vice versa.

Lemma 22 (Multi-step: Sharing within equivalence classes simulates sharing across equivalence classes)**.**
$\forall k \in \mathbb{N}$,
$\mathcal{C}_{init} \nearrow^{0}_{CM} \mathcal{C}_{init} \nearrow^{1}_{CM} \cdots \nearrow^{k}_{CM} \mathcal{C}cm_{k+1}$
implies
$\exists \mathcal{C}tcm_{k+1}$ *s.t.*

$\mathcal{C}_{init} \searrow^{0}_{CM} \mathcal{C}_{init} \searrow^{1}_{CM} \cdots \searrow^{k}_{CM} \mathcal{C}_{tcm_{k+1}}$
and $\mathcal{C}_{cm_{k+1}} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm_{k+1}}$.

Proof. By induction over k and using Single-step: Sharing within equivalence classes simulates sharing across equivalence classes (Lemma 23). □

Lemma 23 (Single-step: Sharing within equivalence classes simulates sharing across equivalence classes)**.**
$\mathcal{C}_{cm} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm}$
and $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm'}$
implies
$\exists \mathcal{C}_{tcm'}$ *s.t.*
$\mathcal{C}_{tcm} \searrow_{CM} \mathcal{C}_{tcm'}$
and $\mathcal{C}_{cm'} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm'}$.

Proof. By inversion on rules for $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm'}$ using Single-step per node: sharing within equivalence classes simulates sharing across equivalence classes (Lemma 24), and applying the rules for $\mathcal{C}_{tcm} \searrow_{CM} \mathcal{C}_{tcm'}$. □

Lemma 24 (Single-step per node: sharing within equivalence classes simulates sharing across equivalence classes)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
and $\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$
implies
$\exists \mathcal{T}_{cm'_\ell}$ *s.t.*
$\mathcal{T}_{cm_\ell} \hookrightarrow \mathcal{T}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm'_\ell} \cdots \mathcal{T}_{cm_N}$.

Proof. By inversion on rules for $\mathcal{S}_{cm_\ell} \hookrightarrow \mathcal{S}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$, using *fireRulesCM* simulates *fireRulesAcrossCM* (Lemma 25) and applying the rules for $\mathcal{T}_{cm_\ell} \hookrightarrow \mathcal{T}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$. □

Lemma 25 (*fireRulesCM* simulates *fireRulesAcrossCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
and $\bar{DQ} \subseteq DQ$
and $fireRulesCM(@\iota_q, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
implies
$\exists \mathcal{L}'_\ell$, $\exists \mathcal{N}'_\ell$ *s.t.*
$fireRulesAcrossCM(@\iota_q, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm_\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof. By induction over length of $\bar{DQ}$, inversion on the rules for $fireRulesCM(@\iota_\ell, \Delta\bar{DQ}, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$, using *fireSingleRuleCM* simulates *fireSingleRuleAcrossCM* (Lemma 26) and applying the rules for *fireRulesAcrossCM*. □

Lemma 26 (*fireSingleRuleCM* simulates *fireSingleRuleAcrossCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{sn_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $fireSingleRuleCM(@\iota_q, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
implies
$\exists \mathcal{L}'_\ell$, $\exists \mathcal{N}'_\ell$ *s.t.*
$fireSingleRuleAcrossCM(@\iota_q, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm_\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm_\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof. By inversion on the rules for $fireSingleRuleCM(@\iota_\ell, \Delta r, u_{cm_\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$, using *compressionCM* simulates *compressionAcrossCM* (Lemma 27) and applying the rules for *fireSingleRuleAcrossCM*. □

Lemma 27 (*compressionCM* simulates *compressionAcrossCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm_\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$

and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
and $\Sigma' \subseteq sel(\Sigma, \Delta r)$
and $compressionCM(@\iota_\ell, \Sigma', \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
implies
$\exists \mathcal{L}'_\ell$, $\exists \mathcal{N}'_\ell$ *s.t.*
and $compressionAcrossCM(@\iota_q, \Sigma', \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof. By induction on the length of Σ', inversion on the rules for $compressionCM(@\iota_\ell, \Sigma', \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$, using *singleCompressionCM* simulates *singleCompressionAcrossCM* (Lemma 28) and applying the rules for *compressionAcrossCM*. □

Lemma 28 (*singleCompressionCM* simulates *singleCompressionAcrossCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
and $\Sigma' \in sel(\Sigma, \Delta r)$
and $\sigma \in \Sigma'$
and $singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
implies
$\exists \mathcal{L}'_\ell$, $\exists \mathcal{N}'_\ell$ *s.t.*
$singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.
Assume that
(1) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
(2) $r \in DQ$
(3) $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
(4) $\Sigma' \in sel(\Sigma, \Delta r)$
(5) $\sigma \in \Sigma'$
(6) $singleCompressionCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$

By the bisimulation relation in (1),
(7) $\Upsilon_\ell \sim\!\sim_{ruleExec} \mathcal{L}_\ell; \mathcal{N}_\ell$
Thus the set of rule provenances in both executions correspond

Case I: $\Gamma(q)[tuple] = \mathsf{event}$**.**

Subcase A: $u_{cm\ell}.\mathsf{createFlag} = Create$**.**
By assumption
The last rule that derived (6) was CM-Create
By inversion we have
(a1) $\Delta r = rID\ \Delta p(@\ell_p, \vec{x}_p) \text{ :- } \Delta q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(a2) $u_{cm\ell} = \langle q(@\iota_q, \vec{t}_q), Create, \mathtt{eID}, \lambda_q \rangle$
(a3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q)$
(a4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(a5) $\forall i \in [1, n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(a6) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(a7) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(a9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \lambda_q{:}3)$
(a10) $u_{cm'_\ell} = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p \rangle$
(a11) $ruleExec_p = \langle \lambda_p, ruleargs, \lambda_q \rangle$
(a12) $\Upsilon'_\ell = \Upsilon_\ell \cup ruleExec_p$
(a13) if $(\sigma(@\ell_p) = @\iota_q)$ then $\mathcal{U}_{cm'_{in}} = [u_{cm'_\ell}], \mathcal{U}_{cm'_{ext}} = []$ else $\mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm'_\ell}]$

We use the above constructs to define:
(a14) $lcm_p \triangleq (\lambda_p, \lambda_q)$
(a15) $ncm_p \triangleq (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_p)$
(a16) $\mathcal{L}'_\ell \triangleq \mathcal{L}_\ell \cup lcm_p$
(a17) $\mathcal{N}'_\ell \triangleq \mathcal{N}_\ell \cup ncm_p$

Using the above constructs we apply CM-ACROSS-CREATE to obtain
(a19) $singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$

By property (1) of the bisimulation relation,
(a20) $\Upsilon_\ell \sim\sim_{ruleExec} \mathcal{L}; \mathcal{N}$
By (a20), (a16), (a17), we apply $\sim\sim_{ruleExec}$-IND and obtain:
(a21) $\Upsilon'_\ell \sim\sim_{ruleExec} \mathcal{L}'; \mathcal{N}'$
We have shown that the rule provenance storage in both executions again relate after the executions take a step.

By (1) and (a21),
(a22) $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$

By (a20) and (a22),
the conclusion holds

Subcase B: $u_{cm\ell}.\mathsf{createFlag} = \mathit{NCreate}$**.**
Since the set of rule provenances is not updated in both executions, the desired conclusion is obvious.

Case II: $\Gamma(q)[tuple] = \mathsf{fast}$**.**

Subcase A: $u_{cm\ell}.\mathsf{createFlag} = \mathit{Create}$**.** Identical argument to Case I, Subcase A.

Subcase B: $u_{cm\ell}.\mathsf{createFlag} = \mathit{NCreate}$**.**
Since the set of rule provenances is not updated in both executions, the desired conclusion is obvious.

□

G.2.4 Online compression sharing storage across *equivalence classes simulates online compression sharing storage* within *equivalence classes*

In this appendix, we show that Online Compression (via sharing storage across equivalence classes) simulates Online Compression (via sharing storage within equivalence classes). We show that given any network configuration $\mathcal{C}_{tcm}$ (where $\mathcal{C}_{tcm} = \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_i} \cdots \mathcal{T}_{cm_N}$) for Online Compression (via sharing storage across equivalence classes), there exists a corresponding network configuration $\mathcal{C}_{cm}$ (where $\mathcal{C}_{cm} = \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_i} \cdots \mathcal{S}_{cm_N}$) for Online Compression (via sharing storage across equivalence classes), such that $\mathcal{C}_{cm} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm}$.

To prove this, for each set of transition rules for Online Compression (via sharing storage across equivalence classes), we state and prove a lemma that shows that these rules have a corresponding counterpart in Online Compression (via sharing storage within equivalence classes). If initially the network configuration for both systems relate, after Online Compression (via sharing storage across equivalence classes) steps to a new configuration, then Online Compression (via sharing storage within equivalence classes) is also able to step to a corresponding new configuration.

We present the necessary lemmas below, but omit most of the proof details as they are similar to those presented in appendix G.1.2. Only the proof of *singleCompressionAcrossCM* simulates *singleCompressionCM* (Lemma 35) differs somewhat lemma that handles the updates of rule provenances. The proof exploits the fact that for every rule provenance element in $\mathcal{C}_{cm}$, there is one corresponding rule provenance link and node in $\mathcal{C}_{tcm}$ and vice versa.

Lemma 29 (Multi-step: sharing across equivalence classes simulates sharing within equivalence classes)**.**
$\forall k \in \mathbb{N}$,
$\mathcal{C}_{init} \searrow^0_{CM} \mathcal{C}_{init} \searrow^1_{CM} \cdots \rightarrow^k_{SN} \mathcal{C}_{cm_{k+1}}$
implies
$\exists \mathcal{C}_{tcm_{k+1}}$ *s.t.*
$\mathcal{C}_{init} \nearrow^0_{CM} \mathcal{C}_{init} \nearrow^1_{CM} \cdots \nearrow^k_{CM} \mathcal{C}_{tcm_{k+1}}$
and $\mathcal{C}_{cm_{k+1}} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm_{k+1}}$.

Proof. By induction over k and using Single-step: sharing across equivalence classes simulates sharing within equivalence classes (Lemma 30). □

Lemma 30 (Single-step: sharing across equivalence classes simulates sharing within equivalence classes)**.**
$\mathcal{C}_{cm} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm}$
and $\mathcal{C}_{tcm} \searrow_{CM} \mathcal{C}_{tcm'}$
implies
$\exists \mathcal{C}_{cm'}$ *s.t.*

$\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm'}$
and $\mathcal{C}_{cm'} \sim\sim_{\mathcal{C}} \mathcal{C}_{tcm'}$.

Proof. By inversion on rules for $\mathcal{C}_{tcm} \searrow_{CM} \mathcal{C}_{tcm'}$ using Single-step per node: sharing across equivalence classes simulates sharing within equivalence classes (Lemma 31), and applying the rules for $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm'}$. □

Lemma 31 (Single-step per node: sharing across equivalence classes simulates sharing within equivalence classes)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cmN}$
and $\mathcal{T}_{cm\ell} \hookrightarrow \mathcal{T}_{cm'\ell}, \mathcal{U}_{cm'_{ext}}$
implies
$\exists \mathcal{S}_{cm'_\ell}$ *s.t.*
$\mathcal{S}_{cm\ell} \hookrightarrow \mathcal{S}_{cm'\ell}, \mathcal{U}_{cm'_{ext}}$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm'_\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cm'_\ell} \cdots \mathcal{T}_{cmN}$.

Proof. By inversion on rules for $\mathcal{T}_{cm\ell} \hookrightarrow \mathcal{T}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$, using *fireRulesAcrossCM* simulates *fireRulesCM* (Lemma 25) and applying the rules for $\mathcal{T}_{cm\ell} \hookrightarrow \mathcal{T}_{cm'_\ell}, \mathcal{U}_{cm'_{ext}}$. □

Lemma 32 (*fireRulesAcrossCM* simulates *fireRulesCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cmN}$
where $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov\ell} \rangle$
and $\mathcal{T}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov\ell} \rangle$
and $\bar{DQ} \subseteq DQ$
and $fireRulesAcrossCM(@\iota_q, \Delta\bar{DQ}, u_{cm\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
implies
$\exists \Upsilon'_\ell$ *s.t.*
$fireRulesAcrossCM(@\iota_q, \Delta\bar{DQ}, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}^{\partial}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}^{\partial}_{cm\ell} \cdots \mathcal{T}_{cmN}$
where $\mathcal{S}^{\partial}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \bar{\mathcal{U}}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov\ell} \rangle$.

Proof.
By induction over length of $\bar{DQ}$,
inversion on the rules for $fireRulesAcrossCM(@\iota_\ell, \Delta\bar{DQ}, u_{cm\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$,
using *fireSingleRuleAcrossCM* simulates *fireSingleRuleCM* (Lemma 33)
and applying the rules for *fireRulesCM*.

□

Lemma 33 (*fireSingleRuleAcrossCM* simulates *fireSingleRuleCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cmN}$
where $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{sn\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov\ell} \rangle$
and $\mathcal{T}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov\ell} \rangle$
and $r \in DQ$
and $fireSingleRuleAcrossCM(@\iota_q, \Delta r, u_{cm\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
implies
$\exists \Upsilon'_\ell$ *s.t.*
$fireSingleRuleCM(@\iota_q, \Delta r, u_{cm\ell}, \mathcal{DB}_\ell, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}^{\partial}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}^{\partial} \cdots \mathcal{T}_{cmN}$
where $\mathcal{S}^{\partial}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov\ell} \rangle$.

Proof.
By inversion on the rules for
$fireSingleRuleAcrossCM(@\iota_\ell, \Delta r, u_{cm\ell}, \mathcal{DB}_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$,
using *compressionAcrossCM* simulates *compressionCM* (Lemma 34)
and applying the rules for *fireSingleRuleCM*.

□

Lemma 34 (*compressionAcrossCM* simulates *compressionCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cm\ell} \cdots \mathcal{S}_{cmN} \sim\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cm\ell} \cdots \mathcal{T}_{cmN}$
where $\mathcal{S}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov\ell} \rangle$
and $\mathcal{T}_{cm\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov\ell} \rangle$
and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
and $\Sigma' \subseteq sel(\Sigma, \Delta r)$

and $compressionAcrossCM(@\iota_\ell, \Sigma', \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
implies
$\exists \Upsilon'_\ell$ *s.t.*
and $compressionAcrossCM(@\iota_q, \Sigma', \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{sn'_{in}}, \mathcal{U}_{sn'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, \mathcal{U}_{cm\ell} \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.
By induction on the length of Σ',
inversion on the rules for $compressionAcrossCM(@\iota_\ell, \Sigma', \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$,
using *singleCompressionAcrossCM* simulates *singleCompressionCM* (Lemma 35)
and applying the rules for *compressionCM*.

□

Lemma 35 (*singleCompressionAcrossCM* simulates *singleCompressionCM*)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
and $r \in DQ$
and $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
and $\Sigma' \in sel(\Sigma, \Delta r)$
and $\sigma \in \Sigma'$
and $singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$
implies
$\exists \mathcal{L}'_\ell$, $\exists \mathcal{N}'_\ell$ *s.t.*
$singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$
and $\mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}^{\partial}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \circ \mathcal{U}_{cm'_{ext}} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}^{\partial}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{sn\ell} :: \bar{\mathcal{U}}_{sn\ell}) \circ \mathcal{U}_{sn'_{in}}, equiSet_\ell, \Upsilon'_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}^{\partial}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, (u_{cm\ell} :: \bar{\mathcal{U}}_{cm\ell}) \circ \mathcal{U}_{cm'_{in}}, equiSet_\ell, \mathcal{L}'_\ell, \mathcal{N}'_\ell, \Upsilon_{prov_\ell} \rangle$.

Proof.
Assume that
(1) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_\ell} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{\mathcal{C}} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_\ell} \cdots \mathcal{T}_{cm_N}$
where $\mathcal{S}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \Upsilon_\ell, \Upsilon_{prov_\ell} \rangle$
and $\mathcal{T}_{cm_\ell} = \langle @\iota_q, DQ, \Gamma, \mathcal{DB}_\ell, \mathcal{E}_\ell, u_{cm\ell} :: \bar{\mathcal{U}}_{cm_\ell}, equiSet_\ell, \mathcal{L}_\ell, \mathcal{N}_\ell, \Upsilon_{prov_\ell} \rangle$
(2) $r \in DQ$
(3) $\Sigma = \rho(\Delta r, q(@\iota_q, \vec{t}_q), \mathcal{DB}_\ell)$
(4) $\Sigma' \in sel(\Sigma, \Delta r)$
(5) $\sigma \in \Sigma'$
(6) $singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm\ell}, \mathcal{L}_\ell, \mathcal{N}_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \mathcal{L}'_\ell, \mathcal{N}'_\ell)$

By the bisimulation relation in (1),
(7) $\Upsilon_\ell \sim\!\sim_{ruleExec} \mathcal{L}_\ell; \mathcal{N}_\ell$
Thus the set of rule provenances in both executions correspond

Case I: $\Gamma(q)[tuple] = \mathsf{event}$**.**

Subcase A: $u_{cm\ell}.\mathsf{createFlag} = Create$**.**
By assumption
The last rule that derived (6) was CM-ACROSS-CREATE
By inversion we have
(a1) $\Delta r = rID\ p(@\ell_p, \vec{x}_p) \text{ :- } q(@\ell_q, \vec{x}_q), b_1(@\ell_q, \vec{x}_{b1}), \cdots, b_n(@\ell_q, \vec{x}_{bn}), \cdots$
(a2) $u_{cm\ell} = \langle q(@\iota_q, \vec{t}_q), Create, \mathtt{eID}, \lambda_q \rangle$
(a3) $q(@\ell_q, \vec{x}_q)\sigma = q(@\iota_q, \vec{t}_q)$
(a4) $\mathsf{dom}(\sigma) = \ell_p \cup \vec{x}_p \cup \ell_q \cup \vec{x}_q \cup \bigcup_{i=1}^{n} \vec{x}_{bi}$
(a5) $\forall i \in [1, n]$, $\mathtt{vID}_i = \text{TUPLEHASH}(b_i(@\ell_q, \vec{x}_{bi})\sigma, \Gamma)$
(a6) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(a9) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \lambda_q{:}3)$
(a10) $u_{cm'_\ell} = \langle p(@\ell_p, \vec{x}_p)\sigma, Create, \mathtt{eID}, \lambda_p \rangle$
(a11) $ncm_p = (\langle @\iota_q, \mathtt{HrID}_p \rangle, ruleargs_p)$
(a12) $\mathcal{N}'_\ell = \mathcal{N}_\ell \cup ncm_p$
(a13) $lcm_p = (\lambda_p, \lambda_q)$
(a14) $\mathcal{L}'_\ell = \mathcal{L}_\ell \cup lcm_p$

(a15) if $\sigma(@\ell_p) = @\iota_q$ then $\mathcal{U}_{cm'_{in}} = [u_{cm'_\ell}], \mathcal{U}_{cm'_{ext}} = []$ else $\mathcal{U}_{cm'_{in}} = [], \mathcal{U}_{cm'_{ext}} = [u_{cm'_\ell}]$

We use the above constructs to define:
(a16) $ruleExec_p \triangleq \langle \lambda_p, ruleargs_p, \lambda_q \rangle$
(a17) $\Upsilon'_\ell = \Upsilon_\ell \cup ruleExec_p$
By definition of the constructs,
(a18) $ruleExec_p \sim\sim_\ell lcm_p :: ncm_p$
By (1) and (a18),
(a19) $\Upsilon'_\ell \sim\sim_\mathcal{C} \mathcal{L}'_\ell; \mathcal{N}'_\ell$
We have shown that the rule provenance storage in both executions again relate after the executions take a step.

Using the above constructs we apply CM-CREATE to obtain
(a20) $singleCompressionAcrossCM(@\iota_q, \sigma, \Delta r, u_{cm_\ell}, \Upsilon_\ell) = (\mathcal{U}_{cm'_{in}}, \mathcal{U}_{cm'_{ext}}, \Upsilon'_\ell)$

By (a19) and (a20)
the conclusion follows

Subcase B: u_{cm_ℓ}.createFlag = *NCreate*.
Since the set of rule provenances is not updated in both executions, the desired conclusion is obvious.

Case II: $\Gamma(q)[tuple]$ = fast.

Subcase A: u_{cm_ℓ}.createFlag = *Create*. The argument is similar to that of Case I, Subcase A.

Subcase B: u_{cm_ℓ}.createFlag = *NCreate*.
Since the set of rule provenances is not updated in both executions, the desired conclusion is obvious.

□

G.3 Proof of Correctness of Compression

We prove the correctness of the online compression algorithm by showing that the distributed provenances maintained in the **ruleExec** and **prov** tables contain the exact same set of provenances of tuples derived by a semi-naïve evaluation. Theorem 3 in Section 5.3 states that we can assemble entries in **ruleExec** and **prov** to reconstruct a provenance tree and vice versa.

Theorem 3 (Correctness of Compression). $\forall n \in \mathbb{N}$ and initial state $\mathcal{C}_{init}$, $\mathcal{C}_{init} \rightarrow^n_{SN} \mathcal{C}_{sn}$ then exists $\mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \rightarrow^n_{CM} \mathcal{C}_{cm}$ and for any derivation tree $tr \in \mathcal{C}_{sn}$, there exists a provenance $\mathcal{P} \in \mathcal{C}_{cm}$ s.t. $tr \sim_d \mathcal{P}$ and for all provenance $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a derivation tree $tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d \mathcal{P}$. And the same is true for the semi-naïve when $\mathcal{C}_{init} \rightarrow^n_{CM} \mathcal{C}_{cm}$.

Theorem 3 is a corollary of Lemma 4, which shows that the semi-naïve execution with online compression algorithm is bisimilar to the semi-naïve execution that stores full derivation trees. The bisimilarity relation relates the distributed compressed provenances and the full derivation provenances in such a way that both store the same set of provenances.

Lemma 4 (Compression Simulates Semi-naïve). $\forall n \in \mathbb{N}$ *given initial state* $\mathcal{C}_{init}$, *and* $\mathcal{C}_{init} \rightarrow^n_{SN} \mathcal{C}_{sn}$ *then* $\exists \mathcal{C}_{cm}$ *s.t.* $\mathcal{C}_{init} \rightarrow^n_{CM} \mathcal{C}_{cm}$ *and* $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_\mathcal{C}}$ $\mathcal{C}_{cm}$ *and vice versa.*

Proof. By Semi-Naïve simulates Online Compression (Lemma 6) and Online Compression simulates Semi-Naïve (Lemma 15).
□

H. CORRECTNESS OF QUERY

We show for both online compression execution that shares storage within equivalence classes and online compression execution that shares storage across equivalent classes, all provenance trees generated by using the semi-naïve evaluation can be queried for, and furthermore the query algorithm will return the correct provenance tree. Because online compression execution may propagate updates out of order, there are situations where rule provenance entries are referred to in a provenance before they are stored. Thus the query algorithm assumes all updates have already been processed.

This section is organized as follows. First we present the query algorithms for both versions of online compression. Next, we define several properties of the provenance that we use in the proof. By Correctness of Compression (Theorem 3) we know there is a bisimulation between semi-naïve evaluation and online compression execution that shares storage within equivalence classes. We use this bisimulation relation to prove correctness of query. Finally, we use the bisimulation relation between the two versions of online compression to see that we can retrieve provenance trees from the network that executed online compression with sharing across equivalence classes.

H.1 Query algorithms

Given a tuple *res* that is an instance of a relation of interest, the query algorithm returns all possible provenance trees for *res*. We present the algorithms for both versions of online compression here.

H.1.1 Sharing storage within *equivalence classes*

We present the query algorithm to retrieve provenances in Figure 31. Function QueryS first checks that all updates in the network have already been processed to ensure that the algorithm is able to retrieve all rule provenances associated with tuple *res*, where *res* is an instance of a relation of interest. The network configuration $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$ stores all rule provenances needed to reconstruct *res*. Each rule provenance for *res* takes the form of *yl*, an ordered list of *ruleExec* elements. The elements in *yl* may be stored at different nodes in the network.

For every instance of a relation of interest derived, the online compression algorithm additionally maintains a tuple provenance *prov* that contains a pointer to the last element of the corresponding list of rule provenance *yl*. ObtainTupleProvS returns all such tuple provenances *prov* associated with *res*. Next, QueryS calls QuerySS, which uses the pointer to the last element of *yl* to return *yl* in its entirety.

Because each element *ruleExec* stores a reference to the previous rule provenance element, QuerySS is able to retrieve every element of *yl* in reverse order. It terminates when the reference to the previous rule provenance is a null pointer, meaning that the final rule provenance retrieved represents the first rule triggered for the execution that derived *res*.

```
1: function QueryS(Q_cm ▷ S_cm1 ··· S_cmN, res, eID)
2:     [prov_1, ··· , prov_m] ← ObtainTupleProvS(Q_cm ▷ S_cm1 ··· S_cmN, res, eID)
3:     ProvSetS = {}
4:     for i ∈ [1, m] do
5:         ⟨@ι_r, res, eID, λ_r⟩ ← prov_i
6:         yl_i ← QuerySS(Q_cm ▷ S_cm1 ··· S_cmN, λ_r)
7:         ProvSetS ← ProvSetS ∪ yl_i
8:     return ProvSetS
9: end function
10:
11: function QuerySS(Q_cm ▷ S_cm1 ··· S_cmN, λ_p)
12:     if λ_p ∈ ⋃_{i=1}^{N} S_cmi.equiSet then
13:         return []
14:     else
15:         ruleExec_p ← Get_RuleExec(⋃_{i=1}^{N} S_cmi.Υ, λ_p)
16:         ⟨λ_p, ruleargs_p, λ_q⟩ ← ruleExec_p
17:         return QuerySS(Q_cm ▷ S_cm1 ··· S_cmN, λ_q) :: ruleExec_p
18: end function
```

Figure 31: Query algorithm for online compression execution that shares storage within equivalence classes

Finally, QueryS returns *ProvSetS*, a set of lists of rule provenances that can be used to recover the provenance trees for *res*. Algorithm CompressedS_to_ProvenanceTree in Figure 32 takes as arguments a rule provenance *yl* in *ProvSetS*, the tuple of interest *res*, the complete set of all materialized tuples in $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$, the mapping of tuples to primary keys Γ, the unique identifier `eID` for the event tuple `eID`, and the DELP program *DQ* and recovers the provenance tree.

```
1: function CompressedS_to_ProvenanceTree(yl :: ruleExec, P, DB, Γ, eID, DQ)
2:     if yl = [] then
3:         ⟨λ_p, ruleargs_p, heq⟩ ← ruleExec
4:         rID :: ι_e :: vID_1 :: ··· :: vID_n ← ruleargs_p
5:         (* DQ[rID] = p(@ℓ_p, x⃗_p) :- e(@ℓ_e, x⃗_e), b_1(@ℓ_e, x⃗_b1), ··· , b_n(@ℓ_e, x⃗_b1) *)
6:         ev ← Recover_Tuples(eID, DB, Γ)
7:         B_1 :: ··· :: B_n ← Recover_Tuples(vID_1 :: ··· :: vID_n, DB, Γ)
8:         return (rID, P, ev, B_1 :: ··· :: B_n)
9:     else
10:        ⟨λ_p, ruleargs_p, λ_q⟩ ← ruleExec_p
11:        rID :: ι_q :: vID_1 :: ··· :: vID_n ← ruleargs_p
12:        B_1 :: ··· :: B_n ← Recover_Tuples(vID_1 :: ··· :: vID_n, DB, Γ)
13:        (* DQ[rID] = p(@ℓ_p, x⃗_p) :- q(@ℓ_q, x⃗_q), b_1(@ℓ_q, x⃗_b1), ··· , b_n(@ℓ_q, x⃗_b1) *)
14:        Q ← Recover_Rule_Trigger(P, B_1 :: ··· :: B_n, rID, DQ)
15:        tr_q ← CompressedS_to_ProvenanceTree(yl, Q, DB, Γ, eID, DQ)
16:        return (rID, P, tr_q:Q, B_1 :: ··· :: B_n)
17: end function
```

Figure 32: Algorithm to recover a provenance tree from an ordered list of rule provenances

H.1.2 Sharing storage across *equivalence classes*

We present the query algorithm to retrieve provenances in Figure 33 below. Function QueryT is almost identical in syntax and semantics to QueryS in appendix H.1.1, except that the network configuration it accepts is for online compression with sharing across equivalence classes, thus the structures for storing rule provenance somewhat differs. Function QueryTT is analogous QuerySS.

```
1: function QueryT(Q_cm ▷ T_cm1 ··· T_cmN, res, eID)
2:     Assert No more updates in Q_cm ▷ T_cm1 ··· T_cmN
3:     [prov_1, ··· , prov_m] ← ObtainTupleProvT(Q_cm ▷ T_cm1 ··· T_cmN, res, eID)
4:     ProvSetT = {}
5:     for i ∈ [1, m] do
6:         ⟨@ι_r, res, eID, λ_r⟩ ← prov_i
7:         ch_i ← QueryTT(Q_cm ▷ T_cm1 ··· T_cmN, λ_r)
8:         ProvSetT ← ProvSetT ∪ {ch_i}
9:
10:    return ProvSetT
11: end function
12:
13: function QueryTT(Q_cm ▷ T_cm1 ··· T_cmN, res, eID)
14:    if λ_p ∈ ⋃_{i=1}^{N} T_cmi.equiSet then
15:        return []
16:    else
17:        id(@ι_q, HrID_p, ♭_p) ← λ_p
18:        lcm ← Get_RuleExec_Link(Q_cm ▷ T_cm1 ··· T_cmN, λ_p)
19:        ncm ← Get_RuleExec_Node(Q_cm ▷ T_cm1 ··· T_cmN, ⟨@ι_q, HrID_p⟩)
20:        (_, λ_q) ← lcm
21:        return QueryTT(Q_cm ▷ T_cm1 ··· T_cmN, λ_q) ⇝ lcm::ncm
22: end function
```

Figure 33: Query algorithm for online compression execution that shares storage across equivalence classes

Algorithm CompressedT_to_ProvenanceTree (Figure 34) recovers the provenance tree corresponding to a list of rule provenances and is analogous to Algorithm CompressedS_to_ProvenanceTree.

```
1: function CompressedT_to_ProvenanceTree(yl ⇝ (lcm :: ncm), P, DB, Γ, eID, DQ)
2:     if yl = [] then
3:         (_, ruleargs_p) ← ncm
4:         rID :: ι_e :: vID_1 :: ··· :: vID_n ← ruleargs_p
5:         (* DQ[rID] = p(@ℓ_p, x⃗_p) :- e(@ℓ_e, x⃗_e), b_1(@ℓ_e, x⃗_b1), ··· , b_n(@ℓ_e, x⃗_b1) *)
6:         ev ← Recover_Tuples(eID, DB, Γ)
7:         B_1 :: ··· :: B_n ← Recover_Tuples(vID_1 :: ··· :: vID_n, DB, Γ)
8:         return (rID, P, ev, B_1 :: ··· :: B_n)
9:     else
10:        (_, ruleargs_p) ← ncm
11:        rID :: ι_q :: vID_1 :: ··· :: vID_n ← ruleargs_p
12:        B_1 :: ··· :: B_n ← Recover_Tuples(vID_1 :: ··· :: vID_n, DB, Γ)
13:        (* DQ[rID] = p(@ℓ_p, x⃗_p) :- q(@ℓ_q, x⃗_q), b_1(@ℓ_q, x⃗_b1), ··· , b_n(@ℓ_q, x⃗_b1) *)
14:        Q ← Recover_Rule_Trigger(P, B_1 :: ··· :: B_n, rID, DQ)
15:        tr_q ← CompressedT_to_ProvenanceTree(yl, Q, DB, Γ, eID, DQ)
16:        return (rID, P, tr_q:Q, B_1 :: ··· :: B_n)
17: end function
```

Figure 34: Algorithm to recover a provenance tree from an ordered list of rule provenances

H.2 Properties of rule provenance

To prove the correctness of query for both forms of online compression, we rely on the fact that every rule provenance element has a unique identifier. We state and prove this in Uniqueness of Rule Provenance Identifier (Lemma 38). In order to prove uniqueness, we defined a well-formness property for rule provenance elements. We say that a rule provenance element *ruleExec* is well-formed when the unique identifier for each element is the hash the unique identifier of the previous rule provenance element generated during program execution. We show that every rule provenance element *ruleExec* (Lemma 36) derived by the online compression execution is well-formed.

H.2.1 Definitions

We define what it means for a rule provenance element *ruleExec* to be well-formed.

Rule WF-HEQ

In the base case only one rule has been fired, thus there is no unique identifier for the previous rule fired. Instead, we record the equivalence hash *heq* as an attribute of the unique identifier for the previous rule.

The rule associated with provenance element $\langle \lambda_p, ruleargs_p, \lambda_e \rangle$ was triggered by an event tuple *ev* with equivalence hash *heq* that joined with some slow-changing tuples $B_1, \cdots, B_n$. To differentiate the unique identifier for the provenance element representing execution of this rule from the provenance element using the same slow-changing tuples $B_1, \cdots, B_n$ but triggered by an event tuple from a different equivalence class, we use *heq* as one of the attributes in the unique identifier.

Rule WF-HashPrev

The rule associated with provenance element $\langle \lambda_p, ruleargs_p, \lambda_q \rangle$ was triggered by a derived fast-changing tuple P that joined with some slow-changing tuples $B_1, \cdots, B_n$. To differentiate the unique identifier for the provenance element representing execution of this rule from the provenance element using the same slow-changing tuples $B_1, \cdots, B_n$ but triggered by an event tuple from a different equivalence class, we hash the unique identifier of the provenance element associated with the previous rule executed.

$\boxed{equiSet \vdash ruleExec\ \mathsf{WF}}$

$$\frac{heq \in equiSet \qquad \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p) \qquad \lambda_p = \mathsf{id}(@\iota_e, \mathtt{HrID}_p, heq) \qquad \lambda_e = \mathsf{id}(\emptyset, \emptyset, heq)}{equiSet \vdash \langle \lambda_p, ruleargs_p, \lambda_e \rangle\ \mathsf{WF}}\ \text{WF-HEQ}$$

$$\frac{equiSet \vdash \langle \lambda_q, \_, \_ \rangle\ \mathsf{WF} \qquad \lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \mathsf{hash}(\lambda_q)) \qquad \mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)}{equiSet \vdash \langle \lambda_p, ruleargs_p, \lambda_q \rangle\ \mathsf{WF}}\ \text{WF-HashPrev}$$

H.2.2 Properties of the provenance when sharing storing within *equivalence classes*

Lemma 36 states that every rule provenance element *ruleExec* stored in $\mathcal{C}_{cm}$ is well-formed. Since $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{C}_{cm}$, every *ruleExec* stored in $\mathcal{C}_{cm}$ corresponds to part of a provenance tree *tr* stored in $\mathcal{C}_{sn}$.

In the base case when only one rule was fired, *ruleExec* corresponds to *tr*. Therefore *ruleExec* is the last element of the list of rule provenances that corresponds to *tr*, so by Well-formness of the last element of a list of rule provenances (Lemma 37) it is well-formed. In the inductive case when multiple rules were fired, there are multiple rule provenances forming a chain *yl* that correspond to *tr*. If *ruleExec* is the last element in *yl*, Well-formness of the last element of a list of rule provenances (Lemma 37) to show that *ruleExec* is well-formed. Otherwise if *ruleExec* is not the last element in *yl*, there must a subtree

tr_s where $tr_s \subseteq tr$ and a subchain yl_s where $yl_s \subseteq yl$ and *ruleExec* is the tail of yl_s, such that tr_s and yl_s correspond. Then again by Well-formness of the last element of a list of rule provenances (Lemma 37) we see that *ruleExec* is well-formed.

Lemma 36 (Well-formness of *ruleExec*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
implies
$\forall ruleExec \in \bigcup_{i=1}^{N} \mathcal{S}_{cmi}.\Upsilon$
$\bigcup_{i=1}^{N} \mathcal{S}_{cmi}.equiSet \vdash ruleExec\ \mathsf{WF}$

Proof.
Assume $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$.
By inversion on the rules for $\boldsymbol{\mathcal{R}_C}$ we have
$\forall i \in [1,N], \mathcal{S}_{sni} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sni}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1,N], \mathcal{S}_{cmi} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cmi}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_U}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1,N],\ \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sni}\ \boldsymbol{\mathcal{R}_U}\ \bigcup_{i=1}^{N} \mathcal{U}_{cmi}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cmi}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{re}}\ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{prov}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$.

Pick any $ruleExec \in \bigcup_{i=1}^{N} \mathcal{S}_{cmi}.\Upsilon$.

By $\mathcal{E}_\delta$,
($\star$) $\exists tr_p{:}P \in \bigcup_{i=1}^{N} \mathcal{M}_i$, $\exists yl \in \bigcup_{i=1}^{N} \Upsilon_i$ s.t.
$\Gamma \vdash tr \sim_d yl$
and $ruleExec \in yl$

We proceed by induction over the structure of $tr_p{:}P$.

Base Case: $tr_p = (rID, P, ev, B_1 :: \cdots :: B_n)$**.**
By ($\star$) and the rules for $\sim_d$,
(b1) $yl = ruleExec$
By (b1),
(b2) $\mathsf{tail}(yl) = ruleExec$
By Well-formness of the last element of a list of rule provenances (Lemma 37),
(b2) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec\ \mathsf{WF}$
The conclusion holds

Inductive Case: $tr_p = (rID, P, tr_q{:}Q, B_1 :: \cdots :: B_n)$**.**

Subcase i: $\mathsf{tail}(yl) = ruleExec$**.**
By Well-formness of the last element of a list of rule provenances (Lemma 37),
(i1) $\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(yl)\ \mathsf{WF}$
By (i1) and since $\mathsf{tail}(yl) = ruleExec$,
(i2) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec\ \mathsf{WF}$
The conclusion holds

Subcase ii: $\mathsf{tail}(yl) \neq ruleExec$**.**
By the assumption that $ruleExec \in yl$,
(ii1) $\exists \hat{yl} \subseteq yl$, $\exists m \in [1, |yl| - 1]$ s.t.
$yl = \hat{yl} :: ruleExec :: ruleExec_1 :: \cdots :: ruleExec_m$
By (ii1) and ($\star$),
(ii2) $\Gamma \vdash tr \sim_d \hat{yl} :: ruleExec :: ruleExec_1 :: \cdots :: ruleExec_m$
By repeated inversion on rule $\sim_d$-IND,
(ii3) $\exists \hat{tr}$ s.t.
$\Gamma \vdash \hat{tr} \sim_d \hat{yl} :: ruleExec$
and $\hat{tr}$ is a subderivation of tr_p
By the Semi-naïve transition rules and (ii3) and since $tr_p{:}P \in \bigcup_{i=1}^{N} \mathcal{M}_i$,
(ii4) $\hat{tr} \in \bigcup_{i=1}^{N} \mathcal{M}_i$
By $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$, (ii3) and (ii4),
we apply Well-formness of the last element of a list of rule provenances (Lemma 37) to obtain
(ii5) $\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(\hat{yl} :: ruleExec)\ \mathsf{WF}$
By (ii5),
(ii6) $\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(ruleExec)\ \mathsf{WF}$

The conclusion holds

□

Well-formness of *ruleExec* (Lemma 36) uses Well-formness of the last element of a list of rule provenances to show that all rule provenances stored by $\mathcal{C}_{cm}$ are well-formed. The proof uses the relation $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$ and induction over the structure of a provenance tree in $\mathcal{C}_{sn}$.

Lemma 37 (Well-formness of the last element of a list of rule provenances)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
implies
$\forall tr \in \bigcup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M},\ \forall yl \subseteq \bigcup_{i=1}^{N} \mathcal{S}_{cm_i}.\Upsilon,$
$\mathcal{S}_{cm_1}.\Gamma \vdash tr \sim_d yl$
implies
$\bigcup_{i=1}^{N} \mathcal{S}_{cm_i}.equiSet \vdash \mathsf{tail}(yl)\ \mathsf{WF}$.

Proof.
Assume $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$.
By inversion on the rules that derived the above,
$\forall i \in [1, N], \mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1, N], \mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$,
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N],\ \Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^{F} \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^{F} \vdash \bigcup_{i=1}^{N} \mathcal{M}_i\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^{F}, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}\ \boldsymbol{\mathcal{R}_{\mathbf{prov}}}\ \bigcup_{i=1}^{N} \Upsilon_{prov_i}$.

Pick any $tr \in \bigcup_{i=1}^{N} \mathcal{M}_i$.

We proceed by induction over the structure of tr.

Base Case: $tr = (rID, p(@\iota_p, \vec{t_p}), e(@\iota_e, \vec{t_e}), b_1(@\iota_e, \vec{t_{b1}}) :: \cdots :: b_n(@\iota_e, \vec{t_{bn}}))$**.**
Pick any $yl \subseteq \bigcup_{i=1}^{N} \mathcal{S}_{cm_i}.\Upsilon$.
Assume $\Gamma \vdash tr \sim_d yl$.
By inversion on the rules for $\sim_d$ we have the following constructs:
(b1) $yl = \mathsf{tail}(yl) = \langle \lambda_p, ruleargs_p, \lambda_q \rangle$
(b2) $heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma)$
(b3) $\forall i \in [1, n],\ \mathtt{vID}_i \textsc{TupleHash}(b_i(@\iota_e, \vec{t_{bi}}), \Gamma)$
(b4) $ruleargs_p = rID :: \iota_e :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(b5) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(b6) $\lambda_p = \mathsf{id}(@\iota_e, \mathtt{HrID}_p, heq)$
By (b2),
(b7) $heq \in \bigcup_{i=1}^{N} equiSet_i$

By the above constructs we apply WF-HEQ to obtain:
(b8) $\bigcup_{i=1}^{N} equiSet_i \vdash \langle \lambda_p, ruleargs_p, \mathsf{id}(\emptyset, \emptyset, heq) \rangle\ \mathsf{WF}$

By (b1) and (b8),
(b9) $\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(yl)\ \mathsf{WF}$

Inductive Case: $tr = (rID, p(@\iota_p, \vec{t_p}), tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\iota_q, \vec{t_{b1}}) :: \cdots :: b_n(@\iota_q, \vec{t_{bn}}))$**.**
Pick any $yl \subseteq \bigcup_{i=1}^{N} \Upsilon_i$.
Assume $\Gamma \vdash tr \sim_d yl$.
By inversion on the rules for $\sim_d$ we have the following constructs:
(i1) $yl = yl_\rho :: ruleExec_q :: ruleExec_p$
where $ruleExec_q = \langle \lambda_q, ruleargs_q, \lambda_\rho \rangle$
and $ruleExec_p = \langle \lambda_p, ruleargs_p, \lambda_q \rangle$
(i2) $\Gamma \vdash tr_q{:}q(@\iota_q, \vec{t_q}) \sim_d yl_\rho :: ruleExec_q$
(i3) $\forall i \in [1, n],\ \mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_e, \vec{t_{bi}}), \Gamma)$
(i4) $ruleargs_p = rID :: \iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(i5) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(i6) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \mathsf{hash}(\lambda_q))$
By the transition rules for Semi-naïve evaluation,

(i7) $tr_q{:}q(@\iota_q, \vec{t}_q) \in \bigcup_{i=1}^{N} \mathcal{M}_i$

Using the bisimulation and (i7), we apply I.H. to obtain:
(i8) $\forall \hat{yl} \subseteq \bigcup_{i=1}^{N} \Upsilon_i$,
$\Gamma \vdash tr_q{:}q(@\iota_q, \vec{t}_q) \sim_d \hat{yl}$
implies
$\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(\hat{yl})\ \mathsf{WF}$
By (i1), (i2) and (i8),
(i9) $\bigcup_{i=1}^{N} equiSet_i \vdash \mathsf{tail}(yl_\rho :: ruleExec_q)\ \mathsf{WF}$

By (i9), (i5), (i6), and (i1),
$\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_p$
By (i1),
$\mathsf{tail}(yl) = \mathsf{tail}(yl_\rho :: ruleExec_q :: ruleExec_p) = ruleExec_p$
By the above and (i9),
the conclusion holds

□

Our online compression algorithm may store the rule provenances for the same execution trace on different nodes in the network. In order to allow for querying of the complete provenance of a tuple, each rule provenance stores the unique identifier of the previous rule provenance derived by the execution. Uniqueness of Rule Provenance Identifier (Lemma 38) shows that our constructs for the unique identifier of a rule provenance element allows us to uniquely identify that element.

Lemma 38 (Uniqueness of Rule Provenance Identifier)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
implies
$\forall ruleExec_A \in \bigcup_{i=1}^{N} \mathcal{S}_{cmi}.\Upsilon$, $\forall ruleExec_B \in \bigcup_{i=1}^{N} \mathcal{S}_{cmi}.\Upsilon$,
$ruleExec_A = \langle \mathsf{id}(@\iota_p, \mathtt{HrID}_p, \flat_p), ruleargs_{pA}, \lambda_{qA} \rangle$
$ruleExec_B = \langle \mathsf{id}(@\iota_p, \mathtt{HrID}_p, \flat_p), ruleargs_{pB}, \lambda_{qB} \rangle$
implies
$ruleargs_{pA} = ruleargs_{pB}$
and $\lambda_{qA} = \lambda_{qB}$

Proof.
Assume that $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$.

Pick any $ruleExec_A, ruleExec_B \in \bigcup_{i=1}^{N} \mathcal{S}_{cmi}.\Upsilon$.
Assume:
$ruleExec_A = \langle \lambda_p, ruleargs_{pA}, \lambda_{qA} \rangle$
and $ruleExec_B = \langle \lambda_p, ruleargs_{pB}, \lambda_{qB} \rangle$.

Our goal is to show that $ruleExec_A = ruleExec_B$.

By Well-formness of *ruleExec* (Lemma 36),
(wfA) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_A\ \mathsf{WF}$
(wfB) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B\ \mathsf{WF}$

We proceed by inversion on (wfA) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_A\ \mathsf{WF}$.

Case WF-HEQ**.**
By assumption
(a1) $\lambda_{qA} = \mathsf{id}(\emptyset, \emptyset, heq_A)$
(a2) $heq_A \in \bigcup_{i=1}^{N} equiSet_i$
(a3) $\mathtt{HrID}_{pA} = \mathsf{hash}(ruleargs_{pA})$
(a4) $\lambda_p = \mathsf{id}(@\iota_e, \mathtt{HrID}_{pA}, heq_A)$
Now by inversion on (wfB) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B\ \mathsf{WF}$, we have the following subcases:

Subcase WF-HEQ**.**
By assumption
(b1) $\lambda_{qB} = \mathsf{id}(\emptyset, \emptyset, heq_B)$
(b2) $heq_B \in \bigcup_{i=1}^{N} equiSet_i$
(b3) $\mathtt{HrID}_{pB} = \mathsf{hash}(ruleargs_{pB})$

(b4) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_{pB}, heq_B)$

By (a4) and (b4),
(b5) $\mathtt{HrID}_{pA} = \mathtt{HrID}_{pB}$
(b6) $heq_A = heq_B$

Using the assumption that there are no collisions in hash:
By (b5) we have
(b7) $ruleargs_{pA} = ruleargs_{pB}$

By (b6), (b1), and (a1),
(b9) $\lambda_{qA} = \lambda_{qB}$

By (b7) and (b9),
the conclusion follows

Subcase WF-HashPrev**.**
By assumption
(b1) $\bigcup_{i=1}^{N} equiSet_i \vdash \langle \lambda_{qB}, \_, \_ \rangle$ WF
(b2) $\mathtt{HrID}_{pB} = \mathsf{hash}(ruleargs_{pB})$
(b3) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_{pB}, \mathsf{hash}(\lambda_{qB}))$

By (b3),
(b4) $\mathtt{HrID}_{pA} = \mathtt{HrID}_{pB}$
(b5) $heq_A = \mathsf{hash}(\lambda_{qB})$

By (b4) and the above constructs,
(b6) $ruleargs_{pA} = ruleargs_{pB}$

By (b1),
(b7) $\lambda_{qB} \notin equiSet$

By (a2) we have
(b8) $heq_A \in \bigcup_{i=1}^{N} equiSet_i$

By (b5), (b7), and (b8),
we have a contradiction
The last rule the derived (wfB) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B$ WF was not WF-HashPrev

Case WF-HashPrev**.**
By assumption
(a1) $\bigcup_{i=1}^{N} equiSet_i \vdash \langle \lambda_{qA}, \_, \_ \rangle$ WF
(a2) $\mathtt{HrID}_{pA} = \mathsf{hash}(ruleargs_{pA})$
(a3) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_{pA}, \mathsf{hash}(\lambda_{qA}))$
Now by inversion on (wfB) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B$ WF, we have the following subcases:

Subcase WF-heq**.**
By assumption
(b1) $\lambda_{qB} = \mathsf{id}(\emptyset, \emptyset, heq_B)$
(b2) $heq_B \in \bigcup_{i=1}^{N} equiSet_i$
(b3) $\mathtt{HrID}_{pB} = \mathsf{hash}(ruleargs_{pB})$
(b4) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_{pB}, heq_B)$

By (b4),
(b5) $\mathtt{HrID}_{pB} = \mathtt{HrID}_{pA}$
(b6) $\mathsf{hash}(\lambda_{qA}) = heq_B$

By (a1) and (b6),
(b7) $heq_B \notin \bigcup_{i=1}^{N} equiSet_i$

By (b2), and (b7),
(b8) we have a contradiction
The last rule the derived (wfB) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B$ WF was not WF-heq.

Subcase WF-HashPrev**.**

By assumption
(b1) $\bigcup_{i=1}^{N} equiSet_i \vdash \langle \lambda_{qB}, \_, \_ \rangle$ WF
(b2) $\texttt{HrID}_{pB} = \mathsf{hash}(ruleargs_{pB})$
(b3) $\lambda_p = \mathsf{id}(@\iota_q, \texttt{HrID}_{pB}, \mathsf{hash}(\lambda_{qB}))$

By (a2) and (b2),
(b4) $\texttt{HrID}_{pA} = \texttt{HrID}_{pB}$
Since we assume there are no collisions in **hash**,
(b5) $ruleargs_{pA} = ruleargs_{pB}$

By (a3) and (b3),
(b6) $\mathsf{hash}(\lambda_{qA}) = \mathsf{hash}(\lambda_{qB})$
Since we assume there are no collisions in **hash** and using (b4),
(b7) $\lambda_{qA} = \lambda_{qB}$

By (b5) and (b7),
the conclusion holds

□

H.2.3 Properties of the provenance when sharing storing across *equivalence classes*

Lemma 39 (Uniqueness of *lcm* and *ncm*)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
and $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
implies
(I) $\forall lcm_A \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{L}$, $\forall lcm_B \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{L}$,
$lcm_A = \langle \lambda_p, \lambda_{qA} \rangle$
and $lcm_B = \langle \lambda_p, \lambda_{qB} \rangle$
implies
$\lambda_{qA} = \lambda_{qB}$
and
(II) $\forall ncm_A \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{N}$, $\forall ncm_B \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{N}$,
$ncm_A = (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pA})$
and $ncm_B = (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pB})$
implies
$ruleargs_{pA} = ruleargs_{pB}$

Proof.
Assume the following:
(A1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
(A2) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\!\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$

By inversion (A2),
(1) $\forall i \in [1, N]$, $\mathcal{S}_{cm_i} \sim\!\sim_{\mathcal{S}} \mathcal{T}_{cm_i}$
By inversion on (1),
(2) $\forall i \in [1, N]$,
$\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
and $\mathcal{T}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{L}_i, \mathcal{N}_i, \Upsilon_{prov_i} \rangle$
and $\Upsilon_i \sim\!\sim_{ruleExec} \mathcal{L}_i, \mathcal{N}_i$

Case (I): Proof that each element storing a parent-child relation has a unique identifier
Pick any $lcm_A \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{L}$.
Pick any $lcm_B \in \bigcup_{i=1} \mathcal{T}_{cm_i}.\mathcal{L}$.

Assume that
(i1) $lcm_A = \langle \lambda_p, \lambda_{qA} \rangle$
(i2) $lcm_B = \langle \lambda_p, \lambda_{qB} \rangle$

By (2),
(i3) $\exists ruleExec_A \in \bigcup_{i=1}^{N} \Upsilon_i$ s.t. $ruleExec_A \sim\!\sim_{\ell} \langle \lambda_p, \lambda_{qA} \rangle$
(i4) $\exists ruleExec_B \in \bigcup_{i=1}^{N} \Upsilon_i$ s.t. $ruleExec_B \sim\!\sim_{\ell} \langle \lambda_p, \lambda_{qB} \rangle$

By inversion on (i3),

(i5) $ruleExec_A = \langle \lambda_p, \_, \lambda_{qA} \rangle$
By inversion on (i4),
(i6) $ruleExec_B = \langle \lambda_p, \_, \lambda_{qB} \rangle$

By Uniqueness of Rule Provenance Identifier (Lemma 38),
$ruleExec_A = ruleExec_B$
By the above
(i7) $\lambda_{qA} = \lambda_{qB}$

By (i7),
(i8) $lcm_A = lcm_B$
The conclusion holds

Case (II): Proof that each element storing the rule provenance arguments has a unique identifier
Pick any $\forall ncm_A \in \bigcup_{i=1} \mathcal{T}_{cmi}.\mathcal{N}$.
Pick any $\forall ncm_B \in \bigcup_{i=1} \mathcal{T}_{cmi}.\mathcal{N}$.

Assume that
(ii1) $ncm_A = (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pA})$
(ii2) $ncm_B = (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pB})$

By (2),
(ii3) $\exists ruleExec_A \in \bigcup_{i=1}^{N} \Upsilon_i$ s.t. $ruleExec_A \sim\sim_\ell (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pA})$
(ii4) $\exists ruleExec_B \in \bigcup_{i=1}^{N} \Upsilon_i$ s.t. $ruleExec_B \sim\sim_\ell (\langle @\iota_q, \texttt{HrID}_p \rangle, ruleargs_{pB})$

By inversion on (ii3),
(ii5) $ruleExec_A = \langle \mathsf{id}(@\iota_q, \texttt{HrID}_p, \_), ruleargs_{pA}, \_ \rangle$
By inversion on (ii4),
(ii6) $ruleExec_B = \langle \mathsf{id}(@\iota_q, \texttt{HrID}_p, \_), ruleargs_{pB}, \_ \rangle$

By (A1) we apply Well-formness of *ruleExec* (Lemma 36) to obtain:
(ii7) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_A$ WF
(ii8) $\bigcup_{i=1}^{N} equiSet_i \vdash ruleExec_B$ WF

By (ii5) and (ii7) and the definition of well-formness
(ii9) $\texttt{HrID}_p = \mathsf{hash}(ruleargs_{pA})$
By (ii6) and (ii8) and the definition of well-formness
(ii10) $\texttt{HrID}_p = \mathsf{hash}(ruleargs_{pB})$

Since we assume no hash collisions, by (ii9) and (ii10):
(ii11) $ruleargs_{pA} = ruleargs_{pB}$

By (ii11) and the definitions in (ii1) and (ii2):
the conclusion holds

□

H.3 Correctness of Query

Our goal is to who that we can recover all possible provenance trees for that tuple from the network configuration for online compression that shares storage *across* equivalence classes. We formalize this notion as Correctness of QueryT (Lemma 42).

We first show that given any tuple that is an instance of a relation of interest, we can recover all possible provenance trees for that tuple from the network configuration for online compression that shares storage *within* equivalence classes (Lemma 40). Next, we use the bisimulation relation between the two versions of online compression execution to show that every provenance returned by QueryS has a corresponding provenance returned by QueryT and vice versa (Lemma 43). We use this to prove Lemma 42.

H.3.1 Sharing storage within *equivalence classes*

We show that given any tuple derived by semi-naïve evaluation, once the online compression execution that started out in the same initial state terminates, then given the network configuration of the online compression execution that shares storage within equivalence classes, QueryS is always able to correctly query for the set of all provenance trees of that tuple.

Correctness of QueryS (Lemma 40).
QueryS shows that given a network configuration for online compression execution with sharing storage within equivalence classes, and there are no more updates to be processed, then for every provenance tree tr_r for an instance of a

relation of interest *res* that is derived by the semi-naïve evaluation, when QueryS is given the network configuration and *res* as arguments, it results a set of rule provenances *ProvSetS*. Every element in *ProvSetS* is an ordered list of rule provenances that can be used to reconstruct a provenance tree for *res*. Furthermore, one of the elements in *ProvSetS* can be used to reconstruct tr_r.

Because QueryS calls QuerySS to retrieve complete provenances, the proof uses Correctness of QuerySS (Lemma 41) to show that given tr_r:*res* relates to a list of rule provenances yl_r ($\Gamma \vdash tr_r$:*res* $\sim_d yl_r$), QuerySS takes in the network configuration for the online compression and a pointer to the *last* rule provenance element in yl_r and returns yl_r in its entirety.

In certain constructs used in QueryS, we write $\mathcal{S}_{cm_1}.\Gamma$ to denote the declaration that maps all relations in the program DQ to a type and its primary keys. Because every state in online compression and semi-naïve evaluation stores the same declaration, we could have chosen to write $\mathcal{S}_{cm_i}.\Gamma$ for $i \in [1, N]$ or $\mathcal{S}_{sn_j}.\Gamma$ for j in $[1, N]$ to denote this declaration as well. However, we write $\mathcal{S}_{cm_1}.\Gamma$ as the network presumably has at least one entity.

Correctness of QuerySS (Lemma 41).

Given that the network configurations for Semi-naïve evaluation and online compression execution relate, and Semi-naïve evaluation stores a provenance tree tr_p for a tuple P, then if tr_p relates to a list of rule provenances yl_p ($\Gamma \vdash tr_p$:$P \sim yl_p$) then QuerySS is able to retrieve yl_p given just the network configuration and the unique identifier of the last element in yl_p.

The proof uses induction over the length of yl_p. In the base case, yl_p has only one rule provenance element, $ruleExec_p$. QuerySS uses the unique identifier of $ruleExec_p$ to retrieve $ruleExec_p$ and returns. Since $yl_p = ruleExec_p :: \mathsf{nil}$, QuerySS has successfully recovered yl_p. For the inductive case, yl_p has form $yl_q :: ruleExec_p$, where yl_q is a non-trivial list of rule provenances corresponding to tr_q:Q, the direct subtree of tr_p:P. QuerySS uses the unique identifier of $ruleExec_p$ to retrieve $ruleExec_p$. We use the induction hypothesis to show that QuerySS is then called with the unique identifier of the last element of yl_q and obtains yl_q. The algorithm returns with $yl_q :: ruleExec_p$.

Lemma 40 (Correctness of QueryS)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
and $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{S}_{cm_i}.\mathcal{U}_{cm} = \emptyset$
implies
$\forall \mathsf{interest}(tr_r\text{:}res) \in \bigcup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}_{prov}$
$\exists ProvSetS$ *s.t.*
QueryS($\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, res, \mathtt{eID}$) = *ProvSetS*
and $\exists yl \in ProvSetS$
$yl \subseteq \bigcup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}_{prov}$
and $\mathcal{S}_{cm1}.\Gamma \vdash tr_r$:*res* $\sim_d yl$
and $\forall \hat{yl} \in ProvSetS \backslash yl$,
$\exists \mathsf{interest}(\hat{tr}_r\text{:}res) \in \bigcup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}_{prov}$ *s.t.*
$\mathcal{S}_{cm1}.\Gamma \vdash \hat{tr}_r \sim_d \hat{yl}$.

Proof.
Assume
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
(2) $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i} = \emptyset$

By inversion on the rule that derived (1):
$\forall i \in [1, N]$, $\mathcal{S}_{sn_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{M}_i, \mathcal{M}_{prov_i} \rangle$
$\forall i \in [1, N]$, $\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{cm_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
$\mathcal{E}_\alpha :: \Gamma \vdash \mathcal{Q}_{sn}$ $\boldsymbol{\mathcal{R}_U}$ $\mathcal{Q}_{cm}$
$\mathcal{E}_\beta :: \forall i \in [1, N]$, $\Gamma \vdash \bigcup_{i=1}^{N} \mathcal{U}_{sn_i}$ $\boldsymbol{\mathcal{R}_U}$ $\bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\gamma :: \mathcal{U}_{cm}{}^F \subseteq \mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{U}_{cm_i}$
$\mathcal{E}_\delta :: \Gamma, DQ, \mathcal{U}_{cm}{}^F \vdash \bigcup_{i=1}^{N} \mathcal{M}_i$ $\boldsymbol{\mathcal{R}_{re}}$ $\bigcup_{i=1}^{N} \Upsilon_i$
$\mathcal{E}_\epsilon :: \Gamma, DQ, \mathcal{U}_{cm}{}^F, \bigcup_{i=1}^{N} \Upsilon_i \vdash \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$ $\boldsymbol{\mathcal{R}_{prov}}$ $\bigcup_{i=1}^{N} \Upsilon_{prov_i}$.

Pick any $\mathsf{interest}(tr_r\text{:}res) \in \bigcup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}_{prov}$
Call QueryS($\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, res, \mathtt{eID}$)

By assumption (2),
the are no updates left to be fired
the assertion on Line 2 passes

By the semantics of Obtain_Result_Prov and $\mathcal{E}_\epsilon$,
(3) $\exists prov_r \in \bigcup_{i=1}^{N} \Upsilon_{prov_i}$ s.t.
$prov_r = \langle @\iota_r, res, \_, \_ \rangle$

and $\Gamma, \bigcup_{i=1}^{N} \Upsilon_i \vdash \mathsf{interest}(tr_r{:}res) \sim_{prov} prov_r$

By inversion on the rules for (3),
(4) $\exists ev$, $\exists \lambda_r$, $\exists yl_r$ s.t.
$ev = \textsc{EventOf}(tr_r{:}res)$
and $\mathtt{eID} = \mathsf{hash}(ev, \Gamma)$
and $prov_r = \langle @\iota_r, res, \mathtt{eID}, \lambda_r \rangle$
and $\Gamma \vdash tr_r{:}res \sim_d yl_r$
and $\mathsf{tail}(yl_p){:}1 = \lambda_r$

By (3) and (4),
the list of tuple provenances $[prov_1, \cdots, prov_m]$ returns by Obtain_Result_Provenance is nontrivial
By a similar reasoning in (3) and (4),
(5) $\forall i \in [1, m]$,
$\exists prov_{ri}$, $\exists ev_i$, $\exists tr_{ri}$, $\exists \lambda_{ri}$, $\exists yl_{ri}$ s.t.
$ev_i = \textsc{EventOf}(tr_{ri}{:}res)$
and $\mathtt{eID}_i = \mathsf{hash}(ev_i, \Gamma)$
and $prov_{ri} = \langle @\iota_{ri}, res, \mathtt{eID}_i, \lambda_{ri} \rangle$
and $\Gamma \vdash tr_{ri}{:}res \sim_d yl_{ri}$
and $\mathsf{tail}(yl_{ri}){:}1 = \lambda_{ri}$

By the rules for Semi-naïve evaluation,
(6) $\forall i \in [1, m]$, $tr_{ri}{:}res \in \bigcup_{i=1}^{N} \Upsilon_{prov_i}$

We use the above constructs to apply Correctness of QuerySS (Lemma 41) and obtain
(7) $\forall i \in [1, m]$,
$\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_{ri}) = tr_{ri}{:}res$
where $\Gamma \vdash tr_{ri}{:}res \sim_d yl_{ri}$
and $\mathsf{tail}(yl_{ri}){:}1 = \lambda_{ri}$

By (5), (7), and the semantics of QueryS,
(8) $\textsc{QueryS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, res, \mathtt{eID})$ terminates and returns *ProvSetS*
(9) $\forall \hat{yl} \in ProvSetS$,
$\exists \hat{tr}_r$ s.t.
$\hat{yl} \subseteq \bigcup_{i=1}^{N} \Upsilon_i$
and $\mathsf{interest}(\hat{tr}_r{:}res) \in \bigcup_{i=1}^{N} \Upsilon_{prov}$
and $\Gamma \vdash \hat{tr}_r{:}res \sim_d \hat{yl}$

By (4), (8), and (9),
the conclusion holds

□

Lemma 41 (Correctness of QuerySS)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
and $tr_p{:}p(@\iota_p, \vec{t_p}) \in \bigcup_{i=1}^{N} \mathcal{S}_{snj}.\mathcal{M}$
and $\Upsilon_p \subseteq \bigcup_{j=1}^{N} \mathcal{S}_{cmj}.\Upsilon$
and $\mathcal{S}_{cm1}.\Gamma \vdash tr_p{:}p(@\iota_p, \vec{t_p}) \sim_d yl_p$
and $\mathsf{tail}(yl_p){:}1 = \lambda_p$
implies
$\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p) = yl_p$

Proof.
Assume:
(1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
(2) $tr_p \in \bigcup_{i=1}^{N} \mathcal{S}_{snj}.\mathcal{M}$
(3) $\Upsilon_p \subseteq \bigcup_{j=1}^{N} \mathcal{S}_{cmj}.\Upsilon$
(4) $\mathcal{S}_{cm1}.\Gamma \vdash tr_p \sim_d yl_p$
(5) $\mathsf{tail}(yl_p){:}1 = \lambda_p$

By (5),
yl_p contains at least one element

We proceed by induction on the length of $|yl_p|$.

Base Case: $|yl_p| = 1$**.**

By assumption
the last rule that derive (4) was $\sim_d$-Base
By inversion on the rule we have the following constructs
(b1) $tr_p = (rID, p(@\iota_p, \vec{t_p}), e(@\iota_e, \vec{t_e}), b_1(@\iota_e, \vec{t}_{b1}) :: \cdots :: b_n(@\iota_e, \vec{t}_{bn}))$
(b2) $heq = \textsc{EquiHash}(e(@\iota_e, \vec{t_e}), \Gamma)$
(b3) $\lambda_e = \mathsf{id}(\emptyset, \emptyset, heq)$
(b3) $\forall i \in [1, n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_e, \vec{t}_{bi}), \Gamma)$
(b4) $ruleargs_p = rID :: \iota_e :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(b5) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(b6) $\lambda_p = \mathsf{id}(@\iota_e, \mathtt{HrID}_p, \mathsf{hash}(heq :: \mathtt{HrID}_p))$
(b7) $yl_p = ruleExec_p = \langle \lambda_p, ruleargs_p, \lambda_e \rangle$

By the semantics of QuerySS and (b6)
when $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p)$ is called
since $\lambda_p{:}3 \notin \bigcup_{i=1}^{N} equiSet_i$,
the **else** branch of the **if-else** statement on Lines 12-17 is taken
By the semantics of QuerySS, (b7), and (b3),
(b8) $\exists ruleExec'_p$ s.t.
$ruleExec'_p = \langle \lambda_p, ruleargs'_p, \lambda'_e \rangle$
and $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p)$ returns $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p) :: ruleExec'_p$

By Uniqueness of Rule Provenance Identifier (Lemma 38),
(b9) $ruleExec'_p = ruleExec_p$

By semantics of QuerySS and (b2)
when $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_e)$ is called,
since $\lambda_e{:}3 \in \bigcup_{i=1}^{N} equiSet_i$,
the **if** branch of the **if-else** statement on Lines 12-17 is taken
By the above,
(b10) the empty list [] is returned

By (b8), (b9), (b10),
(b11) $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p)$ returns $ruleExec_p$

Inductive Case: $|yl_p| = k + 1 \geq 2$**.**

By assumption,
$\exists yl_q$, $\exists ruleExec_p$ s.t.
$yl_p = yl_q :: ruleExec_p$
and $|yl_q| = k \geq 1$
By the above,
the last rule that derived (4) was $\sim_d$-Ind
By inversion on that rule we have the following:
(i1) $tr_p = (rID, p(@\iota_p, \vec{t_p}), tr_q{:}q(@\iota_q, \vec{t_q}), b_1(@\iota_q, \vec{t}_{b1}) :: \cdots :: b_n(@\iota_q, \vec{t}_{bn}))$
(i2) $yl_q = yl_\rho :: ruleExec_q$ where $ruleExec_q = \langle \lambda_q, ruleargs_q, \lambda_\rho \rangle$
(i3) $\Gamma \vdash tr_q{:}q(@\iota_q, \vec{t_q})) \sim_d yl_\rho :: ruleExec_q$ where $ruleExec_q = \langle \lambda_q, ruleargs_q, \lambda_\rho \rangle$
(i4) $\forall i \in [1, n]$, $\mathtt{vID}_i = \textsc{TupleHash}(b_i(@\iota_e, \vec{t}_{bi}), \Gamma)$
(i5) $ruleargs_p = rID :: @\iota_q :: \mathtt{vID}_1 :: \cdots :: \mathtt{vID}_n$
(i6) $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$
(i7) $\flat_p = \mathsf{hash}((\lambda_q{:}3) :: \mathtt{HrID}_p)$
(i8) $\lambda_p = \mathsf{id}(@\iota_q, \mathtt{HrID}_p, \flat_p)$

By (i8) and the semantics of QuerySS
when $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p)$ is called
since $\lambda_p{:}3 \notin \bigcup_{i=1}^{N} equiSet_i$,
the **else** branch of the **if-else** statement on Lines 12-17 is taken
By the above,
(i9) the return value is $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_q) :: ruleExec'_p$
where $ruleExec'_p = \textsc{Get\_RuleExec}(\bigcup_{i=1}^{N} \Upsilon_i, \lambda_p)$

By (i9) and the semantics of Get_RuleExec,

(i10) $ruleExec'_p \in \bigcup_{i=1}^{N} \Upsilon_i$
(i11) $\exists ruleargs'_p, \exists \lambda'_q$ s.t.
$ruleExec'_p = \langle \lambda_p, ruleargs'_p, \lambda'_q \rangle$

By Uniqueness of Rule Provenance Identifier (Lemma 38),
(i12) $ruleExec'_p = ruleExec_p$

Since $|yl_q| = k \geq 1$ and by (i3),
(i13) $tr_q{:}q(@\iota_q, \vec{t}_q)$ is nontrivial and there exists constructs such that
$tr_q = (rID_q, q(@\iota_q, \vec{t}_q), tr_\rho{:}\rho(@\iota_\rho, \vec{t}_\rho), b_{\rho 1}(\iota_\rho, \vec{t}_{\rho 1}) :: \cdots :: b_{\rho m}(\iota_\rho, \vec{t}_{\rho m}))$
By the transition rules for Semi-naïve evaluation,
(i14) $tr_q{:}q(@\iota_q, \vec{t}_q) \in \bigcup_{i=1}^{N} \mathcal{M}_i$

By assumption $\Upsilon_p \subseteq \bigcup_{j=1}^{N} \mathcal{S}_{cmj}.\Upsilon$,
(i15) $yl_\rho :: ruleExec_p \subseteq \bigcup_{j=1}^{N} \mathcal{S}_{cmj}.\Upsilon$

Using (1), (i3), (i14) and (i15) and by I.H.,
(i16) $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_p) = yl_\rho :: ruleExec_q :: ruleExec_p = yl_p$
By (i9), (i12), and (i16),
(i17) $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}, \lambda_q) :: ruleExec'_p$

□

H.3.2 Sharing storage across *equivalence classes*

We show that given any tuple derived by semi-naïve evaluation, once the online compression execution that started out in the same initial state terminates, then given the network configuration of the online compression execution that shares storage across equivalence classes, QueryS is always able to correctly query for the set of all provenance trees of that tuple.

Correctness of QueryT **(Lemma 42).**
This is the key lemma that shows that we can recover all possible provenances for a tuple from the network configuration for online compression that shares storage *across* equivalence classes. The proof relies on the fact that QueryS and QueryT return equivalence sets of provenances as show in QueryS implies QueryT (Lemma 43).

Soundness of QueryT **w.r.t.** QueryS **(Lemma 43).**
This lemma shows that given bisimular network configurations and the same tuple to query for, then QueryS and QueryT will return equivalence sets of provenances given. The proof steps through the implementation of the Algorithms QueryS and QueryT and shows that they perform analogous operations.

Soundness of QueryTT **w.r.t.** QuerySS **(Lemma 44).**
This lemma shows that given bisimular network configurations, the same tuple to query for, and the unique identifier of the last provenance element for the derivation of that tuple, then QuerySS and QueryTT will return corresponding provenances for the query tuple. The proof steps through the implementation of the Algorithms QuerySS and QueryTT and shows that they perform analogous operations.

Lemma 42 (Correctness of QueryT)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
and $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN} \sim\!\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cmN}$
and $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{T}_{cmi}.\mathcal{U}_{cm}$
implies
$\forall \mathsf{interest}(tr_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{S}_{sni}.\mathcal{M}_{prov}$
$\exists ProvSetT$ *s.t.*
$\textsc{QueryT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cmN}, res, \texttt{eID}) = ProvSetT$
and $\exists ch \in ProvSetT$ *s.t.*
$ch \subseteq \bigcup_{i=1}^{N} \mathcal{S}_{sni}.\mathcal{M}_{prov}$
and $\mathcal{T}_{cm1}.\Gamma \vdash tr_r{:}res \sim\!\sim_{ch} ch$
and $\forall \hat{ch} \in ProvSetT,$
$\exists \mathsf{interest}(\hat{tr}_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{S}_{sni}.\mathcal{M}_{prov}$ *s.t.*
$\mathcal{T}_{cm1}.\Gamma \vdash \hat{tr}_r \sim\!\sim_{ch} \hat{ch}$

Proof.
Assume the following:
(A1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$
(A2) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN} \sim\!\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm1} \cdots \mathcal{T}_{cmN}$
(A3) $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{T}_{cmi}.\mathcal{U}_{cm}$

By inversion (A2),
(1) $\forall i \in [1,N]$, $\mathcal{S}_{cm_i} \sim\sim_{\mathcal{S}} \mathcal{T}_{cm_i}$
By inversion on (1),
(2) $\forall i \in [1,N]$,
$\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
and $\mathcal{T}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{L}_i, \mathcal{N}_i, \Upsilon_{prov_i} \rangle$
and $\Upsilon_i \sim\sim_{ruleExec} \mathcal{L}_i, \mathcal{N}_i$

Using (A1) and (2) we apply Correctness of QueryS (Lemma 40) to obtain:
(3) $\forall \mathsf{interest}(tr_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$
$\exists ProvSetS$ s.t.
$\textsc{QueryS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, res, \texttt{eID}) = ProvSetS$
and $\exists yl \in ProvSetS$
$yl \subseteq \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$
and $\Gamma \vdash tr_r{:}res \sim_d yl$
and $\forall \hat{yl} \in ProvSetS \backslash yl$,
$\exists \mathsf{interest}(\hat{tr}_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$ s.t.
$\mathcal{S}_{cm_1}.\Gamma \vdash \hat{tr}_r \sim_d \hat{yl}$

Pick any $\mathsf{interest}(tr_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$.
We apply QueryS implies QueryT (Lemma 43) to obtain:
(4) $\exists ProvSetT$ s.t.
$\forall yl \in ProvSetS$, $\exists ch \in ProvSetT$ s.t. $yl \sim\sim_{ch} ch$
and $\forall ch \in ProvSetT$, $\exists yl \in ProvSetS$ s.t. $yl \sim\sim_{ch} ch$
and $\textsc{QueryT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, res, \texttt{eID}) = ProvSetT$

By (3) and (4),
(5) $\exists ch \in ProvSetT$ s.t.
$ch \subseteq \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$
and $\Gamma \vdash tr_r{:}res \sim\sim_{ch} ch$
and $\forall \hat{ch} \in ProvSetT$,
$\exists \mathsf{interest}(\hat{tr}_r{:}res) \in \bigcup_{i=1}^{N} \mathcal{M}_{prov_i}$ s.t.
$\Gamma \vdash \hat{tr}_r \sim\sim_{ch} \hat{ch}$

□

Lemma 43 (Soundness of QueryT w.r.t. QueryS)**.**
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
and $\textsc{QueryS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, res, \texttt{eID}) = ProvSetS$
implies
$\exists ProvSetT$ *s.t.*
$\forall yl \in ProvSetS$, $\exists ch \in ProvSetT$ *s.t.* $yl \sim\sim_{ch} ch$
and $\forall ch \in ProvSetT$, $\exists yl \in ProvSetS$ *s.t.* $yl \sim\sim_{ch} ch$
and $\textsc{QueryT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, res, \texttt{eID}) = ProvSetT$

Proof.
Assume the following:
(A1) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
(A2) $\textsc{QueryS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, res, \texttt{eID}) = ProvSetS$

By the semantics of QueryS and QueryT,
(1) both call a function (ObtainTupleProvS and ObtainTupleProvT)
respectively that return an identical set of tuple provenances $[prov_1, \cdots, prov_m]$
for the tuple *res* and event identifier `eID`
and $\forall i \in [1,m]$, $prov_i = \langle @\iota_r, res, \texttt{eID}, \lambda_r \rangle$
(2) On Lines 4-7 of both functions, the unique identifier to enables querying for the complete provenance for each $prov_i$ is retrived via QuerySS and QueryTT
(3) On Line 8, both functions return the complete set of rule provenances
that derived tuple *res* given the input event with identifier `eID`.

By (A2) and (2),
(4) For each $prov_i$ in $[prov_1, \cdots, prov_m]$,
$\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, \lambda_r) = yl_i$

and $\textsc{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda_r) = ch_i$
and $\mathcal{T}_{cm_i}.\Gamma \vdash yl_i \sim\sim_{ch} ch_i$

By (3) and (4),
The conclusion follows

□

Lemma 44 (Soundness of QueryTT w.r.t. QuerySS)**.**
$\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
$\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
and $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{T}_{cm_i}.\mathcal{U}_{cm}$
and $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, \lambda_p) = yl$
implies
$\exists ch$ *s.t.*
$\textsc{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda_p) = ch$
and $\mathcal{T}_{cm_1}.\Gamma \vdash yl \sim\sim_{ch} ch$

Proof.
Assume the following:
(A1) $\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn_1} \cdots \mathcal{S}_{sn_N}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}$
(A2) $\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N} \sim\sim_{ch} \mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$
(A3) $\mathcal{Q}_{cm} \cup \bigcup_{i=1}^{N} \mathcal{T}_{cm_i}.\mathcal{U}_{cm}$
(A4) $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, \lambda_p) = yl$

By inversion on (A2),
$\forall i \in [1, N]$, $\mathcal{S}_{sn_i} \sim\sim_{\mathcal{S}} \mathcal{T}_{cm_i}$
By inversion on the above,
($\star$) $\forall i \in [1, N]$,
$\mathcal{S}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \Upsilon_i, \Upsilon_{prov_i} \rangle$
and $\mathcal{T}_{cm_i} = \langle @\iota_i, DQ, \Gamma, \mathcal{DB}_i, \mathcal{E}_i, \mathcal{U}_{sn_i}, equiSet_i, \mathcal{L}_i, \mathcal{N}_i, \Upsilon_{prov_i} \rangle$
and $\Upsilon_i \sim\sim_{ruleExec} \mathcal{L}_i, \mathcal{N}_i$

We proceed by induction on the length of yl.

Base Case: $|yl| = 0$**.**
By assumption
(b1) $yl = []$
By (b1) and the semantics of QuerySS,
$\lambda_r \in \bigcup_{i=1}^{N} equiSet_i$
By the above and the semantics of QueryTT
(b2) $ch = []$
By the rules for $\sim\sim_{ch}$,
(b3) $[] \sim\sim_{ch} []$
By (b2) and (b3),
the conclusion follows

Inductive Case: $|yl| = k + 1$**.**
By assumption,
(i1) $|yl| = k + 1 \geq 1$
(i2) $yl \subseteq \bigcup_{i=1}^{N} \Upsilon_i$
where $\exists \hat{yl}$, $\exists ruleExec_p$ s.t.
$ruleExec_p = \langle \lambda_p, ruleargs_p, \lambda_q \rangle$
and $yl = \hat{yl} :: ruleExec_p$
and $ruleExec_p{:}1 = \lambda_p$
By the semantics of QuerySS,
(i2) the **else** branch of the **if-else** statement on Lines 12-17 of QuerySS was taken
(i3) the function finds $ruleExec'_p$ and returns $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, \lambda_p) :: ruleExec_p$
where $\textsc{QuerySS}(\mathcal{Q}_{cm} \rhd \mathcal{S}_{cm_1} \cdots \mathcal{S}_{cm_N}, \lambda_p) = \hat{yl}$

By (i5), when we call QueryTT with arguments $\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}$ and λ_p,
(i6) the **else** branch of the **if-else** statement on Lines 12-17 of QueryTT is taken
By (i2) and the definition of rule provenances, $ruleExec_p$ consists of the following constructs:
(i7) $ruleExec_p = \langle \lambda_p, ruleargs_p, \lambda_q \rangle = \langle \mathsf{id}(@\iota_q, \mathtt{HrID}_p, b_p), ruleargs_p, \lambda_q \rangle$
where $\mathtt{HrID}_p = \mathsf{hash}(ruleargs_p)$

We use the above to define:
(i8) $lcm \triangleq (\lambda_p, \lambda_q)$
(i9) $ncm \triangleq (\langle @\iota_q, \texttt{HrID}_p, \rangle, \mathit{ruleargs}_p)$
By (i6),
(i10) QueryTT returns $\text{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda'_q) \rightsquigarrow (lcm' :: ncm')$
where $lcm = (\lambda_p, \lambda'_q)$ and $ncm = (\langle @\iota_q, \texttt{HrID}_p, \rangle, \mathit{ruleargs}'_p)$
By Uniqueness of lcm and ncm (Lemma 39),
(i11) $lcm' = lcm$ and $ncm' = ncm$

By (i1) we have
$|\hat{yl}| = k$
Using (A1), (A2), (A3), (i3) and the above,
(i12) $\exists \hat{ch}$ s.t.
$\text{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda_q) = \hat{ch}$
and $\mathcal{T}_{cm_1}.\Gamma \vdash \hat{yl} \sim\!\sim_{ch} \hat{ch}$
By (i11) and (i12),
(i13) $\text{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda_p)$
$= \text{QueryTT}(\mathcal{Q}_{cm} \rhd \mathcal{T}_{cm_1} \cdots \mathcal{T}_{cm_N}, \lambda_q) \rightsquigarrow (lcm :: ncm)$
$= \hat{ch} \rightsquigarrow (lcm :: ncm)$
and $\mathcal{T}_{cm_1}.\Gamma \vdash yl \sim\!\sim_{ch} \hat{ch} \rightsquigarrow (lcm :: ncm)$ where $yl = \hat{yl} :: \mathit{ruleExec}_p$
By (i13)
the conclusion follows

□