# Language-integrated Provenance

Stefan Fehrenbach, James Cheney

*University of Edinburgh*
*10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom*

**Abstract**

Provenance, or information about the origin or derivation of data, is important for assessing the trustworthiness of data and identifying and correcting mistakes. Most prior implementations of data provenance have involved heavyweight modifications to database systems and little attention has been paid to how the provenance data can be used outside such a system. We present extensions to the Links programming language that build on its support for language-integrated query to support provenance queries by rewriting and normalizing monadic comprehensions and extending the type system to distinguish provenance metadata from normal data. The main contribution of this article is to show that the two most common forms of provenance can be implemented efficiently and used safely as a programming language feature with no changes to the database system.

## 1. Introduction

A Web application typically spans at least three different computational models: the server-side program, browser-side HTML or JavaScript, and SQL to execute on the database. Coordinating these layers is a considerable challenge. Recently, programming languages such as Links [14], Hop [34] and Ur/Web [12] have pioneered a *cross-tier* approach to Web programming. The programmer writes a single program, which can be type-checked and analyzed in its own right, but parts of it are executed to run efficiently on the multi-tier Web architecture by translation to HTML, JavaScript and SQL. Cross-tier Web programming builds on *language-integrated query* [30, 33], a technique for safely embedding database queries into programming languages, which has been popularized by Microsoft's LINQ library, which provides language-integrated query for .NET languages such as C# and F#. (The language Links was developed concurrently with Meijer et al.'s work on LINQ; their names are coincidentally similar but they are different systems.)

When something goes wrong in a database-backed Web application, understanding what has gone wrong and how to fix it is also a challenge. Often, the database is the primary "state" of the program, and problems arise when this state becomes inconsistent or contains erroneous data. For example, Figure 1 shows Links code for querying data from a (fictional) Scottish tourism database,

*Email addresses:* `stefan.fehrenbach@ed.ac.uk` (Stefan Fehrenbach), `jcheney@inf.ed.ac.uk` (James Cheney)

```
var agencies = table "Agencies"
 with (name:String, based_in:String, phone:String)
 from db;
var externalTours = table "ExternalTours"
 with (name:String, destination:String, type:String, price:Int)
 from db;
var q1 = query {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone)]
}
```

Figure 1: Links table declarations and example query

| name | phone |
|---|---|
| EdinTours | 412 1200 |
| EdinTours | 412 1200 |
| Burns's | 607 3000 |

Figure 2: Example query results

with the result shown in Figure 2. Suppose one of the phone numbers is incorrect: we might want to know *where* in the source database to find the source of this incorrect data, so that we can correct it. Alternatively, suppose we are curious *why* some data is produced: for example, the result shows EdinTours twice. If we were not expecting these results, e.g. because we believe that EdinTours is a bus tour agency and does not offer boat tours, then we need to see additional input data to understand why they were produced.

Automatic techniques for producing such explanations, often called *provenance*, have been explored extensively in the database literature [16, 5, 25]. Neither conventional nor cross-tier Web programming currently provides direct support for provenance. A number of implementation strategies for efficiently computing provenance for query results have been explored [3, 23, 24], but no prior work considers the interaction of provenance with clients of the database.

We propose *language-integrated provenance*, a new approach to implementing provenance that leverages the benefits of language-integrated query. In this article, we present two instances of this approach, one which computes *where-provenance* showing where in the underlying database a result was copied from, and another which computes *lineage* showing all of the parts of the database that were needed to compute part of the result. Both techniques are implemented by a straightforward source-to-source translation which adjusts the types of query expressions to incorporate provenance information and changes the query behavior to generate and propagate this information. Our approach is implemented in Links, and benefits from its strong support for rewriting queries to efficient SQL equivalents, but the underlying ideas may be applicable to other languages that support language-integrated query, such as F# [38], SML# [31], or Ur/Web [12].

Most prior implementations of provenance involve changes to relational database systems and extensions to the SQL query language, departing from the SQL standard that relational databases implement. To date, none of these proposals have been incorporated into the SQL standard or supported by mainstream database systems. If such extensions are adopted in the future, however, we can simply generate queries that use these extensions in Links. In some of these systems, enabling provenance in a query changes the result type of the query (adding an unpredictable number of columns). Our approach is the first (to the best of our knowledge) to provide type-system support that makes sure that the extra information provided by language-integrated provenance queries is used safely by the client.

Our approach builds on Links's support for queries that construct nested collections [11]. This capability is crucial for lineage, because the lineage of an output record is a *set* of relevant input records. Moreover, our provenance translations can be used with queries that construct nested results. Our approach is also distinctive in allowing fine-grained control over where-provenance. In particular, the programmer can decide whether to enable or disable where-provenance tracking for individual input table fields, and whether to keep or discard provenance for each result field.

We present two simple extensions to Links to support where-provenance and lineage, and give (provably type-preserving) translations from both extensions to plain Links. We have implemented both approaches and experimentally validated them using a synthetic benchmark. Provenance typically slows down query evaluation because more data is manipulated. For where-provenance, our experiments indicate a constant factor overhead of 1.5–2.8. For lineage, the slowdown is between 1.25 and 7.55, in part because evaluating lineage queries usually requires manipulating more data. We also compare Links to Perm [23], a database-integrated provenance system, whose authors report slowdowns of 3–30 for a comparable form of lineage. In our experiments Perm generally outperforms Links but Links is within an order of magnitude.

*Contributions and outline.* Section 2 gives a high-level overview of our approach, illustrated via examples. Section 3 reviews background material on Links upon which we rely. This article makes the following three contributions:

- Definition of the $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ extensions to Links, along with their semantics and provenance correctness properties (Section 4)
- Implementations of $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ by type-preserving translation to plain Links (Section 5)
- Experimental evaluation of the implementations on a number of queries (Section 6)

Related work is discussed in greater detail in Section 7.

This article significantly extends an earlier conference paper [17]. The conference version presented the where-provenance and lineage translations and their implementation and evaluation; this article in addition describes the semantics of Links (Section 3), and proves correctness and type-preservation properties that were not included in the conference paper (Sections 4 and 5).

**Agencies**

| (oid) | name | based_in | phone |
|---|---|---|---|
| 1 | EdinTours | Edinburgh | 412 1200 |
| 2 | Burns's | Glasgow | 607 3000 |

**ExternalTours**

| (oid) | name | destination | type | price in £ |
|---|---|---|---|---|
| 3 | EdinTours | Edinburgh | bus | 20 |
| 4 | EdinTours | Loch Ness | bus | 50 |
| 5 | EdinTours | Loch Ness | boat | 200 |
| 6 | EdinTours | Firth of Forth | boat | 50 |
| 7 | Burns's | Islay | boat | 100 |
| 8 | Burns's | Mallaig | train | 40 |

Figure 3: Example input data

## 2. Overview

In this section we give an overview of our approach, first reviewing necessary background on Links and language-integrated query based on comprehensions, and then showing how provenance can be supported by query rewriting in this framework. We will use a running example of a simple tours database, with some example data shown in Figure 3.

### *2.1. Language-integrated query*

Writing programs that interact with databases can be tricky, because of mismatches between the models of computation and data structures used in databases and those used in conventional programming languages. The default solution (employed by JDBC and other typical database interface libraries) is for the programmer to write queries or other database commands as uninterpreted strings in the host language, and these are sent to the database to be executed. This means that the types and names of fields in the query cannot be checked at compile time and any errors will only be discovered as a result of a run-time crash or exception. More insidiously, failure to adequately sanitize user-provided parameters in queries opens the door to SQL injection attacks [35].

Language-integrated query is a technique for embedding queries into the host programming language so that their types can be checked statically and parameters are automatically sanitized. Broadly, there are two common approaches to language-integrated query. The first approach, which we call *SQL embedding*, adds specialized constructs resembling SQL queries to the host language, so that they can be typechecked and handled correctly by the program. This is the approach taken in C# [30, 33], SML# [31], and Ur/Web [12]. The second approach, which we call *comprehension*, uses monadic comprehensions or related constructs of the host language, and generates queries from such expressions. The comprehension approach builds on foundations for querying databases using comprehensions developed by Buneman et al. [4], and has been adopted in languages such as F# [38] and Links [14] as well as libraries such as Database-Supported Haskell [19].

The advantage of the comprehension approach is that it provides a higher level of abstraction for programmers to write queries, without sacrificing performance.

This advantage is critical to our work, so we will explain it in some detail. For example, the query shown in Figure 1 illustrates Links comprehension syntax. It asks for the names and phone numbers of all agencies having an external tour of type *"boat"*. The keyword **for** performs a comprehension over a table (or other collection), and the **where** keyword imposes a Boolean condition filtering the results. The result of each iteration of the comprehension is a singleton collection containing the record (name = e.name,phone = a.phone).

Monadic comprehensions do not always correspond exactly to SQL queries, but for queries that map flat database tables to flat results, it is possible to normalize these comprehension expressions to a form that is easily translatable to SQL [41]. For example, the following query

```
var q1' = query {
  for (e <-- externalTours)
  where (e.type == "boat")
    for (a <-- agencies)
    where (a.name == e.name)
    [(name = e.name, phone = a.phone)]
}
```

does not directly correspond to a SQL query due to the alternation of **for** and **where** operations; nevertheless, query normalization generates a single equivalent SQL query in which the **where** conditions are both pushed into the SQL query's WHERE clause:

```
SELECT e.name AS name, a.phone AS phone
  FROM ExternalTours e, Agencies a
 WHERE e.type = 'boat' AND a.name = e.name
```

Normalization frees the programmer to write queries in more natural ways, rather than having to fit the query into a pre-defined template expected by SQL.

However, this freedom can also lead to problems, for example if the programmer writes a query-like expression that contains an operation, such as print or regular expression matching, that cannot be performed on the database. In early versions of Links, this could lead to unpredictable performance, because queries would unexpectedly be executed on the server instead of inside the database. The current version uses a type-and-effect system (as described by Cooper [13] and Lindley and Cheney [29]) to track which parts of the program must be executed in the host language and which parts may be executed on the database. Using the **query** keyword above forces the typechecker to check that the code inside the braces will successfully execute on the database.

### *2.2. Higher-order functions and nested query results*

Although comprehension-based language-integrated query may seem (at first glance) to be little more than a notational convenience, it has since been extended to provide even greater flexibility to programmers without sacrificing performance.

The original results on normalization (due to Wong [41]) handle queries over flat input tables and producing flat result tables, and did not allow calling user-defined functions inside queries. Subsequent work has shown how to support higher-order functions [13, 26] and queries that construct nested collections [11]. For example, we can use functions to factor the previous query into reusable components, provided the functions are nonrecursive and only perform operations that are allowed in the database.

```
fun matchingAgencies(name) {
  for (a <-- agencies)
  where (a.name == name)
    [(name = e.name, phone = a.phone)]
}
var q1'' = query {
  for (e <-- externalTours)
  where (e.type == "boat")
    matchingAgencies(e.name)
}
```

Cooper's results show that these queries still normalize to SQL-equivalent queries, and this algorithm is implemented in Links. Similarly, we can write queries whose result type is an arbitrary combination of record and collection types, not just a flat collection of records of base types as supported by SQL:

```
var q2 = query {
  for (a <-- agencies)
    [(name = a.name,
      tours = for (e <-- externalTours)
              where (e.name == a.name)
              [(dest = e.destination, type = e.type)]
}
```

This query produces records whose second tours component is itself a collection — that is, the query result is of the type [(name:String,[(dest:String, type:Type)])] which contains a nested occurrence of the collection type constructor []. SQL does not directly support queries producing such nested results — it requires flat inputs and query results.

Our previous work on *query shredding* [11] gives an algorithm that evaluates queries with nested results efficiently by translation to SQL. Given a query whose return type contains $n$ occurrences of the collection type constructor, query shredding generates $n$ SQL queries that can be evaluated on the database, and constructs the nested result from the resulting tables. This is typically much more efficient than loading the database data into memory and evaluating the query there. Links supports query shredding and we will use it in this article to implement lineage.

Both capabilities, higher-order functions and nested query results, are essential building blocks for our approach to provenance. In what follows, we will use these techniques without further explanation of their implementation. The details are covered in previous papers [13, 29, 11], but are not needed to understand our approach.

### *2.3. Where-provenance and lineage*

As explained in the introduction, provenance tracking for queries has been explored extensively in the database community. We are now in a position to explain how these provenance techniques can be implemented on top of language-integrated query in Links. We review two of the most common forms of provenance, and illustrate our approach using examples; the rest of the article will use similar examples to illustrate our implementation approach.

**Where-provenance** is information about where information in the query result "came from" (or was copied from) in the input. Buneman et al. [5] introduced this idea; our approach is based on a later presentation for the nested relational calculus by Buneman et al. [6]. A common reason for asking for where-provenance is to identify the source of incorrect (or surprising) data in a query result. For example, if a phone number in the result of the example query is incorrect, we might ask for its where-provenance. In our system, this involves modifying the input table declaration and query as follows:

```
var agencies = table "Agencies"
 with (name:String, based_in:String, phone:String)
 where phone prov default
```

The annotation **where** phone **prov default** says to assign phone numbers the "default" provenance annotation of the form (Agencies, phone, i) where $i$ is the object id (oid) of the corresponding row. The field value will be of type **Prov**(**String**); the data value can be accessed using the keyword **data** and the provenance can be accessed using the keyword **prov**, as follows:

```
var q1''' = query {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = data a.phone, p_phone = prov a.phone)]
}
```

Figure 4: Links$^{\mathsf{W}}$ query $q1'''$.

The result of this query is as follows:

| name | phone | p_phone |
|---|---|---|
| EdinTours | 412 1200 | (Agencies,phone,1) |
| EdinTours | 412 1200 | (Agencies,phone,1) |
| Burns's | 607 3000 | (Agencies,phone,2) |

We would like to emphasize one important point about our approach to where-provenance: as illustrated by the above query, we need to change the table definitions to indicate which fields carry provenance, and we also need to annotate the query to indicate where the data or provenance are used. This effort is reasonable because queries are typically small, but alternative strategies, such as automatically annotating all fields, could also be considered.

**Why-provenance** is information that explains "why" a result was produced. In a database query setting, this is usually taken to mean a *justification* or *witness* to the query result, that is, a subset of the input records that includes all of the data needed to generate the result record. Actually, several related forms of why-provenance have been studied [16, 5, 7, 24], however, many of these only make sense for set-valued collections, whereas Links currently supports multiset semantics. In this article, we focus on a simple form of why-provenance called *lineage* which is applicable to either semantics.

Intuitively, the lineage of a record $r$ in the result of a query is a subset $L$ of the records in the underlying database $db$ that "justifies" or "witnesses" the fact that $r$ is in the result of $Q$ on $db$. That is, running $Q$ on the lineage $L$ should produce a result containing $r$, i.e. $r \in Q(L)$. Obviously, this property can be satisfied by many subsets of the input database, including the whole database $db$, and this is part of the reason why there exist several different definitions of why-provenance (for example, to require minimality). We follow the common approach of defining the lineage to be the set of all input database records accessed in the process of producing $r$; this is a safe overapproximation to the minimal lineage, and usually is much smaller than the whole database.

We identify records in input database tables using pairs such as (Agencies,2) where the first component is the table name and the second is the row id, and the lineage of an element of a collection is just a collection of such pairs. (Again, this has the benefit that we can use a single type for references to data in multiple input tables.) Using this representation, the lineage for q1 (Figure 1) is as follows:

| name | phone | lineage |
|---|---|---|
| EdinTours | 412 1200 | [(Agencies,1),(ExternalTours,5)] |
| EdinTours | 412 1200 | [(Agencies,1),(ExternalTours,6)] |
| Burns's | 607 3000 | [(Agencies,2),(ExternalTours,7)] |

In our system, to obtain these results we simply use the keyword **lineage** instead of **query**; for example, for q1 we would simply write:

```
lineage {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone)]
}
```

Links's capabilities for normalizing and efficiently evaluating queries provide the key ingredients needed for computing provenance. For both where-provenance and lineage, we can translate programs using the extensions described above, in a way that both preserves types and ensures that the resulting query expressions can be converted to SQL queries. In the rest of this article, we give the details of these translations and present an experimental evaluation showing that its performance is reasonable.

### *2.4. Pragmatics and limitations*

Most research on provenance in databases has focused on the process of propagating annotations (e.g. source locations) through queries to the output. This article is the first to consider support for provenance at the programming language level. Our attempt to do so has raised some interesting issues that have not been considered in this previous work, such as:

1. Where do the initial provenance annotations come from?
2. What are appropriate correctness criteria in a setting where the underlying program may be updated (by the program or other database users)?
3. Should we also track provenance information for updates, and if so how?

In our approach, we require table declarations to be annotated to indicate how the table's data is annotated with provenance. Thus, we do not assume that the underlying relational database schema contains provenance data, but if such data is available we can use it. However, as we shall see, this complicates matters since we need to be able to handle updates to such tables. We deal with this by translating table references to pairs, with the first component containing the raw table reference for use in updates and the second containing a delayed query expression that produces the initial annotated version of the table for use in queries.

Concerning the second question, we revisit correctness criteria for where-provenance and lineage that have been considered in previous work, and show that similar properties hold for our approach. However, as in previous work, our correctness properties assume that the underlying database is unchanging. This is of course not a realistic assumption: Links includes update operations that can change the database tables, and other database users might concurrently update the data or even change the structure of the data. It is an interesting question (beyond the scope of this paper) how to generalize existing criteria for provenance correctness to this setting.

We mention two additional limitations. First, since Links itself does not yet support grouping and aggregation in queries, our approach does not attempt to handle these features either. This is an important obstacle to be overcome in future work. Likewise, we do not consider the process of tracking provenance for updates to the database, even when the updates are performed by Links. This has been considered by Buneman et al. [6], but in this paper we focus on provenance tracking for queries and leave (language-integrated) provenance tracking for updates for future work.

## 3. Links background

We first review a subset of the Links programming language that includes all of the features relevant to our work; we omit some features (such as effect typing, polymorphism, and concurrency) that are not required for the rest of the article. We also present a simplified operational semantics for Links, omitting detail regarding query normalization and shredding that is presented in more detail in previous work [29, 11]. Appendix A lists notations introduced in this paper, with a brief explanation and reference to their first occurrence.

Figure 5 presents a simplified subset of Links syntax, sufficient for explaining the provenance translations in this article. Types include base types $O$ (such as integers, booleans and strings), table types $\textbf{table}(l_i : A_i)_{i=1}^{n}$, function types $A \texttt{->} B$, record types $(l_i : A_i)_{i=1}^{n}$, and collection types $[A]$. In Links, collection types are treated as multisets inside database queries (reflecting SQL's default multiset semantics), but represented as lists during ordinary execution.

Expressions include standard constructs such as constants, variables, record construction and field projection, conditionals, $n$-ary recursive functions and application. We freely use pair types $(A, B)$ and pair syntax $(M, N)$ and projections $M.1$, $M.2$ etc., which are easily definable using records. Constants $c$ can be functions such as integer addition, equality tests, etc.; their types are collected in a signature $\Sigma$. The signature $\Sigma$ is also a simple model of a database: it maps tables to their contents. In Links we write $\textbf{var}\ x = M; N$ for binding a variable $x$ to the value of $M$ in expression $N$. The semantics of the Links constructs

| | | | |
|---|---|---|---|
| Base types | $O$ | ::= | **Int** \| **Bool** \| **String** |
| Rows | $R$ | ::= | $\cdot \mid R, l : A$ |
| Table types | $T$ | ::= | **table**$(R)$ |
| Types | $A, B$ | ::= | $O \mid T \mid A$ `->` $B \mid (R) \mid [A]$ |
| Contexts | $\Gamma$ | ::= | $\cdot \mid \Gamma, x : A$ |
| Expressions | $L, M, N$ | ::= | $c \mid x \mid (l_i = M_i)_{i=1}^n \mid N.l$ |
| | | \| | **fun** $f(x_i\|_{i=0}^n)\ N \mid N(M_i\|_{i=0}^n)$ |
| | | \| | **var** $x = M; N$ \| **if** $(L)\ \{M\}$ **else** $\{N\}$ |
| | | \| | **query** $\{N\}$ \| **table** $name$ **with** $(l_i : O_i)_{i=1}^n$ |
| | | \| | [] \| $[N]$ \| $N$ `++` $M$ \| **empty**$(M)$ |
| | | \| | **for** $(x$ `<-` $L)\ M$ \| **where**$(M)\ N$ |
| | | \| | **for** $(x$ `<--` $L)\ M$ \| **insert** $L$ **values** $M$ |
| | | \| | **update** $(x$ `<--` $L)$ **where** $M$ **set** $N$ |
| | | \| | **delete** $(x$ `<--` $L)$ **where** $M$ |

Figure 5: Syntax of a subset of Links.

discussed so far is call-by-value. The expression **query** $\{M\}$ introduces a query block, whose content is not evaluated in the usual call-by-value fashion but instead first *normalized* to a form equivalent to an SQL query, and then submitted to the database server. The resulting table (or tables, in the case of a nested query result) are then translated into a Links value. Queries can be constructed using the expressions for the empty collection [], singleton collection $[M]$, and concatenation of collections $M$ ++ $N$. In addition, the comprehension expressions **for**(x <-- $M$) $N$ and **for**(x <- $M$) $L$ allow us to form queries involving iteration over a collection. The difference between the two expressions is that **for**($x$ <-- $M$) expects $M$ to be a table reference, whereas **for**($x$ <- $M$) expects $M$ to be a collection. The expression **where** $(M)\ N$ is equivalent to **if** $(M)\ \{N\}$ **else** $\{[]\}$, and is intended for use in filtering query results. The expression **empty** $(M)$ tests whether the collection produced by $M$ is empty. These comprehension syntax constructs can also be used outside a query block, but they are not guaranteed to be translated to queries in that case. The **insert**, **delete** and **update** expressions perform updates on database tables; they are implemented by direct translation to the analogous SQL update operations.

Figure 6 presents the evaluation judgment $\Sigma, M \rightarrow \Sigma', M'$ for Links expressions. We employ evaluation contexts (following Felleisen and Hieb [18]) $\mathcal{E}$ and define the semantics using several axioms that handle redexes and a single inference rule that shows how to evaluate an expression in which a redex occurs inside an evaluation context. The rule for **update** uses syntactic sugar for record update called **with** for brevity. Most of the rules in Figure 6 are pure in the sense that they have no side-effect on the state of the database. Only the rules for **insert**, **delete** and **update** may change the database state. The rules here present the semantics of Links at a high level, and do not model the exact behavior of query evaluation; instead the **query** $\{M\}$ operation just evaluates to $M$. We assume functions used in database queries and updates are total and have a database equivalent. This is assured by a type and effect system in the full language. Lindley and Cheney [29] present a more detailed model that also

$$
\begin{array}{rcl}
\Sigma, (\textsf{\textbf{fun}}\, f(x_i|_{i=0}^n)\, M)(V_i|_{i=0}^n) & \longrightarrow & \Sigma, M[f := \textsf{\textbf{fun}}\, f(x_i)\, M, x_i := V_i] \\
\Sigma, \textsf{\textbf{var}}\, x = V; M & \longrightarrow & \Sigma, M[x := V] \\
\Sigma, (l_i = V_i)_{i=1}^n.l_k & \longrightarrow & \Sigma, V_k \\
\Sigma, \textsf{\textbf{if}}\, (\textsf{\textbf{true}})\, M\, \textsf{\textbf{else}}\, N & \longrightarrow & \Sigma, M \\
\Sigma, \textsf{\textbf{if}}\, (\textsf{\textbf{false}})\, M\, \textsf{\textbf{else}}\, N & \longrightarrow & \Sigma, N \\
\Sigma, \textsf{\textbf{query}}\, M & \longrightarrow & \Sigma, M \\
\Sigma, \textsf{\textbf{empty}}(\texttt{[]}) & \longrightarrow & \Sigma, \textsf{\textbf{true}} \\
\Sigma, \textsf{\textbf{empty}}(V) & \longrightarrow & \Sigma, \textsf{\textbf{false}} \quad \text{iff } V \neq \texttt{[]} \\
\Sigma, \textsf{\textbf{for}}\, (x \texttt{ <- } \texttt{[]})\, M & \longrightarrow & \Sigma, \texttt{[]} \\
\Sigma, \textsf{\textbf{for}}\, (x \texttt{ <- } [V])\, M & \longrightarrow & \Sigma, M[x := V] \\
\Sigma, \textsf{\textbf{for}}\, (x \texttt{ <- } V +\!\!+ W)\, M & \longrightarrow & \Sigma, (\textsf{\textbf{for}}\, (x \texttt{ <- } V)\, M) +\!\!+ (\textsf{\textbf{for}}\, (x \texttt{ <- } W)\, M) \\
\Sigma, \textsf{\textbf{for}}\, (x \texttt{ <-- } \textsf{\textbf{table}}\, n)\, M & \longrightarrow & \Sigma, \textsf{\textbf{for}}\, (x \texttt{ <- } \Sigma(n))\, M \\
\Sigma, \textsf{\textbf{insert}}\, (\textsf{\textbf{table}}\, t)\, \textsf{\textbf{values}}\, V & \longrightarrow & \Sigma[t \mapsto \Sigma(t) +\!\!+ V], ()
\end{array}
$$

$$
\frac{\Sigma' = \Sigma[t \mapsto [X \in \Sigma(t) | \Sigma, M[x := X] \longrightarrow^* \Sigma, \textsf{\textbf{false}}]]}{\Sigma, \textsf{\textbf{delete}}\, (x \texttt{ <-- } \textsf{\textbf{table}}\, t)\, \textsf{\textbf{where}}\, M \longrightarrow \Sigma', ()}
$$

$$
\frac{\Sigma' = \Sigma[t \mapsto [u(X) | X \in \Sigma(t)]] \quad u(X) = \begin{cases} (X\, \textsf{\textbf{with}}\, l_i = V_i) & \text{if } M[x := X] \longrightarrow^* \textsf{\textbf{true}} \\ & \quad \text{and } N_i[x := X] \longrightarrow^* V_i \\ X & \text{otherwise} \end{cases}}{\Sigma, \textsf{\textbf{update}}\, (x \texttt{ <-- } \textsf{\textbf{table}}\, t)\, \textsf{\textbf{where}}\, M\, \textsf{\textbf{set}}\, (l_i = N_i)_{i=1}^n \longrightarrow \Sigma', ()}
$$

$$
\frac{\Sigma, M \longrightarrow \Sigma', M'}{\Sigma, \mathcal{E}[M] \longrightarrow \Sigma', \mathcal{E}[M']}
$$

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [] \mid \mathcal{E}(M_1, \ldots, M_n) \mid V(V_1, \ldots, V_{i-1}, \mathcal{E}, M_{i+1}, \ldots, M_n) \\
& \mid & (l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_i = \mathcal{E}, l_{i+1} = M_{i+1}, \ldots, l_n = M_n) \mid \mathcal{E}.l \\
& \mid & \textsf{\textbf{if}}\, (\mathcal{E})\, M\, \textsf{\textbf{else}}\, N \\
& \mid & \textsf{\textbf{empty}}(\mathcal{E}) \\
& \mid & [\mathcal{E}] \mid \mathcal{E} +\!\!+ M \mid V +\!\!+ \mathcal{E} \\
& \mid & \textsf{\textbf{for}}\, (x \texttt{ <- } \mathcal{E})\, M \mid \textsf{\textbf{for}}\, (x \texttt{ <-- } \mathcal{E})\, M \\
& \mid & \textsf{\textbf{insert}}\, (\mathcal{E})\, M \mid \textsf{\textbf{insert}}\, (\textsf{\textbf{table}}\, n)\, \mathcal{E} \\
& \mid & \textsf{\textbf{update}}\, (x \texttt{ <-- } \mathcal{E})\, \textsf{\textbf{where}}\, M\, \textsf{\textbf{set}}\, (l_i = N_i)_{i=1}^n \\
& \mid & \textsf{\textbf{delete}}\, (x \texttt{ <-- } \mathcal{E})\, \textsf{\textbf{where}}\, M
\end{array}
$$

Figure 6: Semantics of Links.

shows how flat Links queries are normalized and evaluated externally using SQL and Cheney et al. [11] shows how nested queries are implemented.

The type system (again a simplification of the full system) is illustrated in Figure 7. Many rules are standard; we assume a typing signature $\Sigma$ mapping constants and primitive operations to their types. The rule for **query** $\{M\}$ refers to an auxiliary judgment $A :: \mathsf{QType}$ that essentially checks that $A$ is a valid query result type, meaning that it is constructed using base types and collection or record type constructors only:

$$\frac{}{O :: \mathsf{QType}} \qquad \frac{[A_i :: \mathsf{QType}]_{i=1}^n}{(l_i : A_i)_{i=1}^n :: \mathsf{QType}} \qquad \frac{A :: \mathsf{QType}}{[A] :: \mathsf{QType}}$$

Similarly, the $R :: \mathsf{BaseRow}$ judgment ensures that the types used in a row are all base types:

$$\frac{}{\cdot :: \mathsf{BaseRow}} \qquad \frac{R :: \mathsf{BaseRow}}{R, l : O :: \mathsf{BaseRow}}$$

The full Links type system also checks that the body $M$ uses only features available on the database (and only calls functions that satisfy the same restriction). The rules for other query operations are straightforward, and similar to those for monadic comprehensions in other systems. Finally, the rules for updates (insert, update, and delete) are also mildly simplified; in the full system, the conditions and update expressions are required to be database-executable operations. Lindley and Cheney [29] present a more complete formalization of Links's type system that soundly characterizes the intended run-time behavior.

The core language of Links we are using is a simplification of the full language in several respects. Links includes a number of features (e.g. recursive datatypes, XML literals, client/server annotations, and concurrency features) that are important parts of its Web programming capabilities but not needed to explain our contribution. Links also uses a type-and-effect system to determine whether the code inside a **query** block is translatable to SQL, and which functions can be called safely from query blocks. We use a simplified version of Links's type system that leaves out these effects and does not deal with polymorphism. Our implementation does handle these features, with some limitations discussed later.

## 4. Extending Links with provenance

In this paper we follow a well-explored approach to modeling provenance by propagating *annotations* of various kinds. Roughly speaking, the idea is to interpret a query using a nonstandard semantics over data with additional annotations on fields or records. The nonstandard semantics propagates annotations from the input to the output in a way that is intended to convey useful information about how the results were derived from the inputs; sometimes the semantics is proved correct with respect to some specification of the intended meaning. This idea dates to Wang and Madnick's *polygen* model [40], and is adopted in much subsequent work on provenance in databases (see [7] for a survey).

In this section we describe two extensions of Links: $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ which provide language support for where-provenance and lineage, respectively. For both languages, we discuss language design, syntax, semantics, type system, and

$$\textsc{Const}\quad \frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \textsc{Var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Record}\quad \frac{\Gamma \vdash M_i : A_i \quad (i \in \{1,\ldots,n\})}{\Gamma \vdash (l_i \;\texttt{=}\; M_i)_{i=1}^n : (l_i : A_i)_{i=1}^n} \qquad \textsc{Projection}\quad \frac{\Gamma \vdash M : (l_i : A_i)_{i=1}^n}{\Gamma \vdash M.l_k : A_k}$$

$$\textsc{Fun}\quad \frac{\Gamma, [x_i : A_i]_{i=1}^n \vdash M : B}{\Gamma \vdash \textbf{fun}\ (x_i|_{i=1}^n)\{M\} : (A_i|_{i=1}^n)\ \texttt{->}\ B}$$

$$\textsc{App}\quad \frac{\Gamma \vdash M : (A_i|_{i=1}^n)\ \texttt{->}\ B \qquad \Gamma \vdash N_i : A_i \quad (i \in \{1,\ldots,n\})}{\Gamma \vdash M(N_i|_{i=1}^n) : B}$$

$$\textsc{Var}\quad \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \textbf{var}\ x\ = M; N : B} \qquad \textsc{Query}\quad \frac{\Gamma \vdash M : [A] \qquad A :: \mathsf{QType}}{\Gamma \vdash \textbf{query}\ \{M\} : [A]}$$

$$\textsc{Empty}\quad \frac{\Gamma \vdash M : [A]}{\Gamma \vdash \textbf{empty}(M) : \textbf{Bool}} \qquad \textsc{Table}\quad \frac{R :: \mathsf{BaseRow}}{\Gamma \vdash \textbf{table}\ n\ \textbf{with}\ (R) : \textbf{table}(R)} \qquad \textsc{Empty-List}\quad \frac{}{\Gamma \vdash [] : [A]}$$

$$\textsc{List}\quad \frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : [A]} \qquad \textsc{Concat}\quad \frac{\Gamma \vdash M : [A] \qquad \Gamma \vdash N : [A]}{\Gamma \vdash M\ \texttt{++}\ N : [A]} \qquad \textsc{For-List}\quad \frac{\Gamma \vdash L : [A] \qquad \Gamma, x : A \vdash M : [B]}{\Gamma \vdash \textbf{for}\ (x\ \texttt{<-}\ L)\ M : [B]}$$

$$\textsc{Where}\quad \frac{\Gamma \vdash M : \mathsf{Bool} \qquad \Gamma \vdash N : [B]}{\Gamma \vdash \textbf{where}\ (M)\ N : [B]} \qquad \textsc{For-Table}\quad \frac{\Gamma \vdash L : \textbf{table}(R) \qquad \Gamma, x : (R) \vdash M : [B]}{\Gamma \vdash \textbf{for}\ (x\ \texttt{<--}\ L)\ M : [B]}$$

$$\textsc{Insert}\quad \frac{\Gamma \vdash L : \textbf{table}(R) \qquad \Gamma \vdash M : [(R)]}{\Gamma \vdash \textbf{insert}\ L\ \textbf{values}\ M : ()}$$

$$\textsc{Update}\quad \frac{\Gamma \vdash L : \textbf{table}(R) \qquad \Gamma, x : (R) \vdash M : \textbf{Bool} \qquad \Gamma, x : (R) \vdash N : (R)}{\Gamma \vdash \textbf{update}\ (x\ \texttt{<--}\ L)\ \textbf{where}\ M\ \textbf{set}\ N : ()}$$

$$\textsc{Delete}\quad \frac{\Gamma \vdash L : \textbf{table}(R) \qquad \Gamma, x : (R) \vdash M : \textbf{Bool}}{\Gamma \vdash \textbf{delete}\ (x\ \texttt{<--}\ L)\ \textbf{where}\ M : ()}$$

Figure 7: Typing rules for Links.

most importantly, how provenance annotations are propagated. We discuss how to provide initial annotations for $\mathsf{Links}^{\mathsf{W}}$ here, and in Section 5 for $\mathsf{Links}^{\mathsf{L}}$. For both languages, the correctness theorems are only concerned with the faithful propagation of annotations, not what the annotations actually are.

### *4.1.* $\mathsf{Links}^{\mathsf{W}}$

$\mathsf{Links}^{\mathsf{W}}$ extends $\mathsf{Links}$ with language support for computing the where-provenance of database queries. The syntax shown in Figure 5 is extended as follows:

$$\begin{array}{rcl} V & ::= & \cdots \mid V^c \\ O & ::= & \cdots \mid \mathbf{Prov}(O) \\ L, M, N & ::= & \cdots \mid \mathbf{data}\ M \mid \mathbf{prov}\ M \mid \mathbf{table}\ n\ \mathbf{with}\ (R)\ \mathbf{where}\ S \\ S & ::= & \cdot \mid S, l\ \mathbf{prov}\ s \\ s & ::= & \mathbf{default} \mid M \end{array}$$

Values $V$ can be annotated with an element $c$ of some sufficiently large set of distinguishable atomic annotations, often called *colors*. We will use where-provenance triples for colors. That is, an annotation consists of a triple $(R, f, i)$ where $R$ is the source table name, $f$ is the field name, and $i$ is the row identifier. We introduce the type constructor $\mathbf{Prov}(O)$, where $O$ is a type argument of base type. We treat $\mathbf{Prov}(O)$ itself as a base type, so that it can be used as part of a table type. (This is needed for initializing provenance as explained below.) Values of type $\mathbf{Prov}(O)$ are annotated values $V^c$, where the annotation consists of a triple $(R, f, i)$ where $R$ is the source table name, $f$ is the field name, and $i$ is the row identifier. For example, $42^{("QA", "a", 23)}$ represents the answer 42 which was copied from row 23, column $\mathsf{a}$, of table $\mathsf{QA}$. The syntax above allows arbitrary values to be annotated; however, the type system will only permit values of base type to be annotated. Annotated values are not available in source programs; only the $\mathsf{Links}^{\mathsf{W}}$ runtime can construct annotated values.

$$\begin{array}{rcl} \Sigma, \mathbf{prov}\, V^c & \longrightarrow & \Sigma, c \\ \Sigma, \mathbf{data}\, V^c & \longrightarrow & \Sigma, V \end{array}$$

$$\mathcal{E} \quad ::= \quad \cdots \mid \mathbf{prov}\,\mathcal{E} \mid \mathbf{data}\,\mathcal{E}$$

Figure 8: Additional evaluation and context rules for $\mathsf{Links}^{\mathsf{W}}$.

We add two additional keywords **prov** and **data** to extract from an annotated value the provenance annotation and the value itself, respectively. We extend the semantics from Figure 6 with rules for these keywords as seen in Figure 8.

Only the $\mathsf{Links}^{\mathsf{L}}$ runtime can create annotated values, and it only annotates database values. We allow programmers to indicate which columns in a database table should carry annotations and give some control over what the annotations themselves are. To this end, we extend the syntax of table expressions to allow a list of *provenance initialization specifications* $l$ **prov** $s$. A specification $s$ is either the keyword **default** or an expression $M$ which is expected to be of type $(\overline{l_i : O_i})$ `->` $(\mathbf{String}, \mathbf{String}, \mathbf{Int})$. This way we have three different kinds of columns: plain columns without annotations; columns with *default* where-provenance where the annotation will be the table name, column name, and the

$$\textsc{Prov}\quad \frac{\Gamma \vdash M : \textsf{\textbf{Prov}}(A)}{\Gamma \vdash \textsf{\textbf{prov}}\ M : (\textsf{\textbf{String}}, \textsf{\textbf{String}}, \textsf{\textbf{Int}})}
\qquad
\textsc{Data}\quad \frac{\Gamma \vdash M : \textsf{\textbf{Prov}}(A)}{\Gamma \vdash \textsf{\textbf{data}}\ M : A}$$

$$\textsc{Table}\quad \frac{R :: \textsf{BaseRow} \qquad \Gamma \vdash S : \textsf{ProvSpec}(R)}{\Gamma \vdash \textsf{\textbf{table}}\ n\ \textsf{\textbf{with}}\ (R)\ \textsf{\textbf{where}}\ S : \textsf{\textbf{table}}(R \triangleright S)}
\qquad
\textsc{Insert}\quad \frac{\Gamma \vdash L : \textsf{\textbf{table}}(R) \qquad \Gamma \vdash M : [(\lfloor R \rfloor)]}{\Gamma \vdash \textsf{\textbf{insert}}\ L\ \textsf{\textbf{values}}\ M : ()}$$

$$\textsc{Update}\quad \frac{\Gamma \vdash L : \textsf{\textbf{table}}(R) \qquad \Gamma, x : (\lfloor R \rfloor) \vdash M : \textsf{\textbf{Bool}} \qquad \Gamma, x : (\lfloor R \rfloor) \vdash N : (R)}{\Gamma \vdash \textsf{\textbf{update}}\ (x \texttt{<--}\ L)\ \textsf{\textbf{where}}\ M\ \textsf{\textbf{set}}\ N : ()}$$

$$\textsc{Delete}\quad \frac{\Gamma \vdash L : \textsf{\textbf{table}}(R) \qquad \Gamma, x : (\lfloor R \rfloor) \vdash M : \textsf{\textbf{Bool}}}{\Gamma \vdash \textsf{\textbf{delete}}\ (x \texttt{<--}\ L)\ \textsf{\textbf{where}}\ M : ()}$$

$$\frac{}{\Gamma \vdash \cdot : \textsf{ProvSpec}(R)}
\qquad
\frac{\Gamma \vdash S : \textsf{ProvSpec}(R)}{\Gamma \vdash S, l\ \textsf{\textbf{prov default}} : \textsf{ProvSpec}(R)}$$

$$\frac{\Gamma \vdash S : \textsf{ProvSpec}(R) \qquad \Gamma \vdash M : (R)\ \texttt{->}\ (\textsf{\textbf{String}}, \textsf{\textbf{String}}, \textsf{\textbf{Int}})}{\Gamma \vdash S, l\ \textsf{\textbf{prov}}\ M : \textsf{ProvSpec}(R)}$$

Figure 9: Additional typing rules for $\textsf{Links}^{\textsf{W}}$.

$$\begin{array}{rcl}
\lfloor O \rfloor & = & O \\
\lfloor \textsf{\textbf{Prov}}(A) \rfloor & = & \lfloor A \rfloor \\
\lfloor (l_i : A_i)_{i=1}^n \rfloor & = & (l_i : \lfloor A_i \rfloor)_{i=1}^n \\
R \triangleright \cdot & = & R \\
(R, l : O) \triangleright (S, l\ \textsf{\textbf{prov}}\ s) & = & (R \triangleright S), l : \textsf{\textbf{Prov}}(O)
\end{array}$$

Figure 10: $\textsf{Links}^{\textsf{W}}$ type erasure and augmentation.

row's oid; and columns with annotations that are computed by some user-defined function that takes the table row as input.

Default where-provenance can be understood as user-defined where-provenance with a compiler-generated function of the form **fun** (r) { (T, C, r.oid) } where T and C are replaced by the table and column name, respectively. For example, if we added default where-provenance to the phone field of the Agencies table, we would execute the following function on every row, to obtain the phone numbers provenance: **fun** (a) { (*"Agencies"*, *"phone"*, a.oid) }.

The typing rules for the new constructs of $\mathsf{Links}^{\mathsf{W}}$ are shown in Figure 9. These rules employ an auxiliary judgment $\Gamma \vdash S : \mathsf{ProvSpec}(R)$, meaning that in context $\Gamma$, the provenance specification $S$ is valid with respect to record type $R$. As suggested by the typing rule, the **prov** keyword extracts the provenance from a value of type $\mathbf{Prov}(A)$, and **data** extracts its data, the $A$-value. The most complex rule is that for the **table** construct.

The rules make use of an *erasure* operation $\lfloor R \rfloor$ that takes a record or base type and replaces all occurrences of $\mathbf{Prov}(A)$ with $A$. The rule for typing table references also uses an auxiliary operation $R \triangleright S$ that defines the type of the provenance view of a table whose fields are described by $R$ and whose provenance specification is $S$. As for ordinary tables, we check that the fields are of base type. These operations are defined in Figure 10.

The following proofs and definitions are based on previous work by Buneman et al. [6] in the context of nested relational algebra. The main correctness property of where-provenance is that annotations on values are correctly propagated. It should not be the case that we construct annotated values out of thin air. For the propagation behavior to be correct, it does not matter what the annotations are or where they come from. Buneman et al. discuss some other interesting properties which do not hold in our language. In their work, annotations are completely abstract, and queries have no way to inspect them. Therefore, they can show that queries are invariant under recoloring of the input. $\mathsf{Links}^{\mathsf{W}}$ has the **prov** keyword to inspect provenance, therefore we cannot expect the same to hold here. However, we speculate that a similar property holds for sufficiently polymorphic functions.

We assume a context $\Sigma$ where values inside tables are annotated with colors. We do not make any assumptions about these colors. However, they are particularly useful when they are distinct. In the case of distinct annotations on the input, we can look at the output and trace back annotated values to their source (assuming evaluation does not conjure up new annotated values out of thin air). In Figure 11 we define the function $cso_\Sigma$ for finding all *colored subobjects* of a $\mathsf{Links}^{\mathsf{W}}$ term. This function allows us to find the annotations in the program and state that we do not invent any during evaluation. Thus, if we start with a distinctly annotated database and no annotated constants, we can then guarantee that all annotated values in the result of evaluation come, without modification, directly from the database. Theorem 2 formally states this intuition of evaluation not inventing annotated values.

We first show a helpful lemma: the colored subobjects of a term substituted into an evaluation context $\mathcal{E}[M]$ can be obtained by considering the evaluation context $\mathcal{E}$ and term $M$ separately, instead. We extend $cso_\Sigma(-)$ to operate on evaluation contexts in the obvious way.

$$
\begin{array}{lcl}
cso_\Sigma(V^a) & = & \{V^a\} \cup cso_\Sigma(V) \\
cso_\Sigma(c) & = & \emptyset \\
cso_\Sigma(\texttt{[]}) & = & \emptyset \\
cso_\Sigma(\texttt{[}M\texttt{]}) & = & cso_\Sigma(M) \\
cso_\Sigma(M \mathbin{+\!\!+} N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma((l_i = M_i)_{i=1}^n) & = & \bigcup_{i=1}^n cso_\Sigma(M_i) \\
cso_\Sigma(M.l) & = & cso_\Sigma(M) \\
cso_\Sigma(\textsf{fun}\, f(x_i|_{i=1}^n)\, M) & = & cso_\Sigma(M) \\
cso_\Sigma(M(N_i|_{i=1}^n)) & = & cso_\Sigma(M) \cup \bigcup_{i=1}^n cso_\Sigma(N_i) \\
cso_\Sigma(\textsf{var}\, x = M; N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\textsf{if}\,(L)\, M\, \textsf{else}\, N) & = & cso_\Sigma(L) \cup cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\textsf{query}\, M) & = & cso_\Sigma(M) \\
cso_\Sigma(\textsf{table}\, n) & = & cso_\Sigma(\Sigma(n)) \\
cso_\Sigma(\textsf{empty}(M)) & = & cso_\Sigma(M) \\
cso_\Sigma(\textsf{for}\,(x \,\texttt{<-}\, M)\, N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\textsf{for}\,(x \,\texttt{<--}\, M)\, N) & = & cso_\Sigma(M) \cup cso_\Sigma(N)
\end{array}
$$

Figure 11: Colored subobjects in $\textsf{Links}^{\textsf{W}}$ expressions.

**Lemma 1.** Given evaluation context $\mathcal{E}$ and term $M$, we have:

$$cso_\Sigma(\mathcal{E}[M]) = cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M)$$

*Proof.* Proof by induction on the structure of the evaluation context. In the case for $\mathcal{E} = []$ we take the colored subobjects of a hole to be the empty set. The other cases are straightforward.

□

**Theorem 2** (Correctness of where-provenance)**.** *Let $M$ and $N$ be $\textsf{Links}^{\textsf{W}}$ terms, and let $\Sigma$ be a context that provides annotated table rows. We have:*

$$\Sigma, M \longrightarrow \Sigma, N \Rightarrow cso_\Sigma(N) \subseteq cso_\Sigma(M)$$

*Proof.* Proof by induction on the derivation of the evaluation relation $\longrightarrow$. We show some representative cases here, the full proof is in Appendix B.1.

- Case $\textsf{for}\,(x \,\texttt{<-}\, \texttt{[]})\, M \longrightarrow \texttt{[]}$: $cso_\Sigma(\texttt{[]}) = \emptyset \subseteq cso_\Sigma(\textsf{for}\,(x \,\texttt{<-}\, \texttt{[]})\, M)$
- Case $\textsf{for}\,(x \,\texttt{<-}\, \texttt{[}V\texttt{]})\, M \longrightarrow M[x := V]$:

$$
\begin{aligned}
cso_\Sigma(M[x := V]) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(V) \\
&= cso_\Sigma(\textsf{for}\,(x \,\texttt{<-}\, \texttt{[}V\texttt{]})\, M)
\end{aligned}
$$

- Case $\textsf{for}\,(x \,\texttt{<-}\, V \mathbin{+\!\!+} W)\, M \longrightarrow (\textsf{for}\,(x \,\texttt{<-}\, V)\, M) \mathbin{+\!\!+} (\textsf{for}\,(x \,\texttt{<-}\, W)\, M)$:

$$
\begin{aligned}
cso_\Sigma(\textsf{for}\,(x \,\texttt{<-}\, V \mathbin{+\!\!+} W)\, M) &= cso_\Sigma(V \mathbin{+\!\!+} W) \cup cso_\Sigma(M) \\
&= cso_\Sigma(V) \cup cso_\Sigma(W) \cup cso_\Sigma(M) \\
&= cso_\Sigma((\textsf{for}\,(x \,\texttt{<-}\, V)\, M) \mathbin{+\!\!+} (\textsf{for}\,(x \,\texttt{<-}\, W)\, M))
\end{aligned}
$$

- Case $M \longrightarrow M' \Rightarrow \mathcal{E}[M] \longrightarrow \mathcal{E}[M']$ (evaluation step inside a context):

$$
\begin{aligned}
cso_\Sigma(\mathcal{E}[M']) &= cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M') && \text{Lemma 1} \\
&\subseteq cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M) && \text{IH} \\
&= cso_\Sigma(\mathcal{E}[M]) && \text{Lemma 1}
\end{aligned}
$$

$$\begin{array}{rcl}
\textbf{Lin}(A) & = & (\mathsf{data} : A, \mathsf{prov} : [(\textbf{String}, \textbf{Int})]) \\
\mathfrak{L}[\![O]\!] & = & O \\
\mathfrak{L}[\![A \;\texttt{->}\; B]\!] & = & \mathfrak{L}[\![A]\!] \;\texttt{->}\; \mathfrak{L}[\![B]\!] \\
\mathfrak{L}[\![(l_i : A_i)_{i=1}^n]\!] & = & (l_i : \mathfrak{L}[\![A_i]\!])_{i=1}^n \\
\mathfrak{L}[\![[A]]\!] & = & [\textbf{Lin}(\mathfrak{L}[\![A]\!])] \\
\mathfrak{L}[\![\textbf{table}(R)]\!] & = & \mathfrak{L}[\![[(R)]]\!]
\end{array}$$

Figure 12: Lineage type translation

$$\textsc{Lineage}\quad \frac{\Gamma \vdash M : [A] \qquad A :: \mathsf{QType}}{\Gamma \vdash \textbf{lineage}\ \{M\} : \mathfrak{L}[\![[A]]\!]}$$

Figure 13: Additional typing rule for $\mathsf{Links}^{\mathsf{L}}$

□

*4.2. Lineage*

$\mathsf{Links}^{\mathsf{L}}$ adds the keyword **lineage** to Links. Like the keyword **query**, it is followed by a block of code that will be translated into SQL and executed on the database. The **query** keyword only affects where and how the evaluation takes place. The result is the same as if database tables were lists in memory. The **lineage** keyword also triggers translation of the following code block into SQL. However, the query is rewritten to not only compute the result, but every row of the result is annotated with its lineage. The syntax is extended as follows:

$$L, M, N \quad ::= \quad \cdots \mid \textbf{lineage}\{M\}$$

The expression **lineage** $\{M\}$ is similar to **query** $\{M\}$, in that $M$ must be an expression that can be executed on the database (that is, terminating and side-effect free; this is checked by Links's effect type system just as for **query** $\{\mathsf{M}\}$). If $M$ has type $[A]$ (which must be an appropriate query result type) then the type of the result of **lineage** $\{M\}$ will be $\mathfrak{L}[\![[A]]\!]$, where $\mathfrak{L}[\![-]\!]$ is a type translation that adjusts the types of collections $[A]$ to allow for lineage, as shown in Figures 12 and 13.

A **lineage** block evaluates in one step to its result, as can be seen in Figure 14. The result is determined by a second evaluation relation that is only used "inside" lineage blocks: $\longrightarrow_{\mathsf{L}}$. The language which $\longrightarrow_{\mathsf{L}}$ operates on is $\mathsf{Links}^{\mathsf{L}}$, except that list values are replaced by a variant of lists, $\hat{L}$, where every list element is annotated with a set of colors:

$$\begin{array}{rcl}
V & ::= & \cdots \mid \hat{L} \\
\hat{L} & ::= & \texttt{[]} \mid \texttt{[}V\texttt{]}^a \mid \hat{L} +\!\!+ \hat{L} \\
M & ::= & \cdots \mid M^{\cup b}
\end{array}$$

$$\frac{\hat{\Sigma}, \mathit{annotate}(M) \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}, \hat{L}}{\Sigma, \textsf{lineage}\, M \longrightarrow \Sigma, \mathit{a2d}(\hat{L})}$$

$$\begin{aligned}
\mathit{annotate}(\texttt{[]}) &= \texttt{[]} \\
\mathit{annotate}(\texttt{[}V\texttt{]}) &= \texttt{[}\mathit{annotate}(V)\texttt{]}^{\emptyset} \\
\mathit{annotate}(V \mathbin{+\!\!+} W) &= \mathit{annotate}(V) \mathbin{+\!\!+} \mathit{annotate}(W)
\end{aligned}$$

$$\begin{aligned}
\mathit{a2d}(\texttt{[]}) &= \texttt{[]} \\
\mathit{a2d}(\texttt{[}V\texttt{]}^{\{a_1,\ldots,a_n\}}) &= \texttt{[}(\mathsf{data} = \mathit{a2d}(V), \mathsf{prov} = \texttt{[}a_1, \ldots, a_n\texttt{]})\texttt{]} \\
\mathit{a2d}(V \mathbin{+\!\!+} W) &= \mathit{a2d}(V) \mathbin{+\!\!+} \mathit{a2d}(W)
\end{aligned}$$

Figure 14: $\mathsf{Links}^{\mathsf{L}}$ semantics.

Note how the set of annotations $a$ is on the singleton list constructor, not the actual element value as you might expect. We use annotations to track lineage, which describes *why* the value, or row, is in the result. Lineage is not concerned with what the value actually is.

We represent lineage as a list of rows in the database and identify rows by their table name and row number. Every occurrence of the list type constructor in the type of a lineage query result is replaced by a list of records of data and its provenance. For example, if a **query** block has type [Bool], the result of the same code in a **lineage** block has type [(data: Bool, prov: [(String, Int)])].

There are two functions for going from $\mathsf{Links}^{\mathsf{L}}$ values to annotated values used inside **lineage** blocks, and back. The first function is *annotate*, which recursively annotates $\mathsf{Links}^{\mathsf{L}}$ lists with empty lineage annotations. We assume an extension of this function to non-list values and arbitrary $\mathsf{Links}^{\mathsf{L}}$ terms in the obvious way. Only rows in database tables will have nonempty lineage annotations, provided by an extended context $\hat{\Sigma}$. The second function is $\mathit{a2d}$, which recursively transforms annotated lists into plain data $\mathsf{Links}^{\mathsf{L}}$ lists. Nonlist values are traversed in the obvious way. Every annotated list element will be transformed into a record with data and prov fields. The prov field will hold the lineage annotations, a set of colors, as a list. Here we assume that colors are $\mathsf{Links}^{\mathsf{L}}$ values. In practice they will be pairs of table name and row number; in theory we could use anything and define one more function to go from color to $\mathsf{Links}^{\mathsf{L}}$ value.

Evaluation inside **lineage** blocks is almost the same as evaluation outside. A **lineage** block is similar to a **query** block in that it can contain only pure, nonrecursive functions, and no database updates. We do not support empty inside lineage blocks, because it can lead to nonmonotonic queries. Figure 15 shows the evaluation rules. The major differences from regular evaluation are in the treatment of **for** comprehensions and the new syntax $M^{\cup b}$. A table comprehension takes the table values from an annotated signature $\hat{\Sigma}$, which maps tables to lists with lineage annotations. A **for** comprehension over a singleton list adds the singleton's annotation to all of the elements in the output list. For this use alone we introduce the new type of expression $M^{\cup b}$. It takes a term and a set of annotations, evaluates the term to a list value, and adds the annotations. This is not syntax intended to be used by the programmer.

$$
\begin{aligned}
\hat{\Sigma}, \texttt{[]}^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, \texttt{[]} \\
\hat{\Sigma}, (\texttt{[}V\texttt{]}^{a})^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, \texttt{[}V\texttt{]}^{a \cup b} \\
\hat{\Sigma}, (V +\!\!+ W)^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V^{\cup b} +\!\!+ W^{\cup b} \\
\hat{\Sigma}, (\mathbf{fun}\, f(x_i|_{i=0}^{n})\, M)(V_i|_{i=0}^{n}) &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x_i := V_i]_{i=0}^{n} \\
\hat{\Sigma}, \mathbf{var}\, x = V; M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x := V] \\
\hat{\Sigma}, \mathbf{for}\, (x \texttt{<-}\ \texttt{[]})\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, \texttt{[]} \\
\hat{\Sigma}, \mathbf{for}\, (x \texttt{<-}\ \texttt{[}V\texttt{]}^{a})\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, (M[x := V])^{\cup a} \\
\hat{\Sigma}, \mathbf{for}\, (x \texttt{<-}\ V +\!\!+ W)\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, (\mathbf{for}\, (x \texttt{<-}\ V)\, M) +\!\!+ \mathbf{for}\, (x \texttt{<-}\ W)\, M \\
\hat{\Sigma}, \mathbf{for}\, (x \texttt{<--}\ \mathbf{table}\, t)\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, \mathbf{for}\, (x \texttt{<-}\ \hat{\Sigma}(t))\, M \\
\hat{\Sigma}, \mathbf{query}(V) &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V \\
\hat{\Sigma}, \mathbf{if}(\mathbf{true})\, M\, \mathbf{else}\, N &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M \\
\hat{\Sigma}, \mathbf{if}(\mathbf{false})\, M\, \mathbf{else}\, N &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, N \\
\hat{\Sigma}, (l_i = V_i)_{i=1}^{n}.l_k &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V_k
\end{aligned}
$$

$$
\mathcal{E} \quad ::= \quad \cdots \mid \mathcal{E}^{\cup b}
$$

Figure 15: Propagation of lineage annotations.

Lineage of a query result tells us which elements of the input were responsible for each element of the output to exist. If we run the same query again, but on only that part of the input that was mentioned in the lineage annotations, we should get the same output. Nonmonotonic queries, that is queries that use aggregations, emptiness tests, or set difference, cause issues here: For example consider the query that selects everything from table $a$ if table $b$ is empty. Every row in the result would be annotated with a corresponding row in $a$. One would also need to record somehow the fact that $b$ was empty. We could annotate whole tables in addition to individual rows, but this would complicate the annotation model. This is the approach taken in the work on dependency provenance [8] which is similar to lineage but extends to nonmonotonic queries. For this work, we chose to only consider monotonic queries.

In order to state the lineage correctness property formally, we need three auxiliary definitions from Figure 16. We only show the most relevant cases here, but extend both functions to the entire language in the obvious way. The full definitions can be found in Appendix B.2. The function $\|\cdot\|$ collects all lineage annotations mentioned in a value and is extended to $\mathsf{Links}^{\mathsf{L}}$ terms. The function $\cdot|_b$ restricts values, in particular list elements, to those annotated with a subset of annotations $b$. We extend this to $\mathsf{Links}^{\mathsf{L}}$ terms in the obvious way and to annotated contexts such that tables mentioned in a restricted context $\hat{\Sigma}|_b$ do not contain rows which are not in $b$. Note that this function always preserves list literals and values originating in the surrounding program because those are annotated with empty lineage. Finally we have the recursive sublist relation $\sqsubseteq$. For example `[(a = [2])]` $\sqsubseteq$ `[(a = [1]), (a = [2, 3])]`.

Suppose a monotonic $\mathsf{Links}^{\mathsf{L}}$ query $q$ evaluates, inside a lineage block, to an

$$\begin{aligned}
\|\,\texttt{[}M\texttt{]}^{a}\| &= a \cup \|M\| \\
\|\,\texttt{[]}\,\| &= \emptyset \\
\|M \mathbin{+\!\!+} N\| &= \|M\| \cup \|N\| \\
\|M^{\cup b}\| &= b \cup \|M\| \\
\|\textsf{table}\, t\| &= \|\hat{\Sigma}(t)\| \\
\|\textsf{for}\,(x \,\texttt{<-}\, M)\, N\| &= \|M\| \cup \|N\|
\end{aligned}$$

$$\begin{aligned}
\texttt{[}M\texttt{]}^{a}|_b &= \begin{cases} \texttt{[}M|_b\texttt{]}^{a} & \text{if } a \subseteq b \\ \texttt{[]} & \text{otherwise} \end{cases} \\
\texttt{[]}\,|_b &= \texttt{[]} \\
(M \mathbin{+\!\!+} N)|_b &= M|_b \mathbin{+\!\!+} N|_b \\
M^{\cup a}|_b &= \begin{cases} (M|_b)^{\cup a} & \text{if } a \subseteq b \\ \texttt{[]} & \text{otherwise} \end{cases} \\
\textsf{table}\, t|_b &= \textsf{table}\, t \\
(\textsf{for}\,(x \,\texttt{<-}\, M)\, N)|_b &= \textsf{for}\,(x \,\texttt{<-}\, M|_b)\, N|_b
\end{aligned}$$

$$\frac{}{V \sqsubseteq V} \qquad \frac{}{\texttt{[]} \sqsubseteq L} \qquad \frac{V \sqsubseteq V'}{\texttt{[}V\texttt{]}^{b} \sqsubseteq \texttt{[}V'\texttt{]}^{b}} \qquad \frac{V \sqsubseteq V' \qquad W \sqsubseteq W'}{V \mathbin{+\!\!+} W \sqsubseteq V' \mathbin{+\!\!+} W'}$$

$$\frac{\forall 1 \leq i \leq n : \qquad l_i = l'_i \qquad V_i \sqsubseteq V'_i}{(l_i = V_i)_{i=1}^{n} \sqsubseteq (l'_i = V'_i)_{i=1}^{n}}$$

Figure 16: Auxiliary definitions to collect lineage, restrict values, and find sublists.

annotated value $\hat{v}$ in a context $\hat{\Sigma}$. For every part $\hat{p}$ of the value $\hat{v}$ we can obtain a smaller context $\hat{\Sigma}|_{\|\hat{p}\|}$ by erasing all values from the original context $\hat{\Sigma}$ which are not mentioned in $\hat{p}$. The lineage annotations are correct if every part $\hat{p} \sqsubseteq \hat{v}$ of the output $\hat{v}$ is also a part of the output $\hat{v}'$ obtained by evaluating the same query $q$ in the restricted context $\hat{\Sigma}|_{\|\hat{p}\|}$.

**Theorem 3.** *Given monotonic terms $M$ and $N$, a context $\hat{\Sigma}$, and a set of annotations $c$, we have*

$$\hat{\Sigma}, M \longrightarrow_{\mathsf{L}} \hat{\Sigma}, N \quad \Rightarrow \quad M|_c = N|_c \quad \vee \quad \hat{\Sigma}|_c, M|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, N|_c$$

*Proof.* By induction on the evaluation relation $\longrightarrow_{\mathsf{L}}$. We need the alternative $M|_c = N|_c$ because sometimes restriction can yield the empty list, on both sides, in which case there is no evaluation step to be made. The two interesting cases are the singleton **for** comprehension, which introduces $M^{\cup a}$, and adding annotations to a singleton list, which eliminates $M^{\cup a}$.

Case $\hat{\Sigma}, \textsf{for}\,(x \,\texttt{<-}\, \texttt{[}V\texttt{]}^{a})\, M \longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x := V]^{\cup a}$:
We have two cases, depending on $c$. If $a \subseteq c$ then $(\textsf{for}\,(x \,\texttt{<-}\, \texttt{[}V\texttt{]}^{a})\, M)|_c =$

$\mathbf{for}\,(x \texttt{<-} \texttt{[}V|_c\texttt{]}^a)\,(M|_c)$ and therefore

$$\hat{\Sigma}|_c, \mathbf{for}\,(x \texttt{<-} \texttt{[}V|_c\texttt{]}^a)\,(M|_c) \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M|_c[x := V|_c])^{\cup a}$$

Furthermore, we have $(M|_c[x := V|_c])^{\cup a} = ((M[x := V])|_c)^{\cup a}$, which can be shown by induction, but only states that $\cdot|_c$ is well-behaved with respect to substitution, and $((M[x := V])|_c)^{\cup a} = (M[x := V])^{\cup a}|_c$ by definition of $M^{\cup a}|_c$ in the case that $a \subseteq c$, and therefore

$$\hat{\Sigma}|_c, (\mathbf{for}\,(x \texttt{<-} \texttt{[}V\texttt{]}^a)\,M)|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M[x := V]^{\cup a})|_c$$

Otherwise $a \not\subseteq c$ and on the left hand side we have

$$(\mathbf{for}\,(x \texttt{<-} \texttt{[}V\texttt{]}^a)\,M)|_c = \mathbf{for}\,(x \texttt{<-} (\texttt{[}V\texttt{]}^a)|_c)\,(M|_c) = \mathbf{for}\,(x \texttt{<-} \texttt{[]})\,(M|_c)$$

which evaluates to the empty list:

$$\hat{\Sigma}|_c, \mathbf{for}\,(x \texttt{<-} \texttt{[]})\,(M|_c) \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, \texttt{[]}$$

Since $(M[x := V]^{\cup a})|_c = \texttt{[]}$ we can conclude that

$$\hat{\Sigma}|_c, (\mathbf{for}\,(x \texttt{<-} \texttt{[}V\texttt{]}^a)\,M)|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M[x := V]^{\cup a})|_c$$

Case $\hat{\Sigma}, (\texttt{[}V\texttt{]}^b)^{\cup a} \longrightarrow_{\mathsf{L}} \hat{\Sigma}, \texttt{[}V\texttt{]}^{a \cup b}$:
Depending on $c$ we, again, have two cases. If $a \subseteq c$ then $(\texttt{[}V\texttt{]}^b)^{\cup a}|_c = (\texttt{[}V\texttt{]}^b|_c)^{\cup a}$. Now, if $b \subseteq c$ then $\texttt{[}V\texttt{]}^b|_c = \texttt{[}V|_c\texttt{]}^b$ and we have an evaluation step $\hat{\Sigma}|_c, (\texttt{[}V|_c\texttt{]}^b)^{\cup a} \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, \texttt{[}V|_c\texttt{]}^{a \cup b}$ where the term on the right hand side is equal to $\texttt{[}V\texttt{]}^{a \cup b}|_c$. Otherwise, $b \not\subseteq c$ and $\texttt{[}V\texttt{]}^b|_c = \texttt{[]}$ but on the right hand side we also have $\texttt{[}V\texttt{]}^{a \cup b}|_c = \texttt{[]}$. In other words, by restricting with $c$ we get the same value on both sides. We reach the same conclusion in the case that $a \not\subseteq c$. □

**Corollary 4.** By repeated application of Theorem 3 we have

$$\hat{\Sigma}, M \longrightarrow^j_{\mathsf{L}} \hat{\Sigma}, N \quad \Rightarrow \quad \hat{\Sigma}|_c, M|_c \longrightarrow^k_{\mathsf{L}} \hat{\Sigma}|_c, N|_c$$

where $j, k \in \mathbb{N}$ and $k \leq j$.

**Lemma 5.** Given a value $\hat{v}$ and a subvalue $\hat{p} \sqsubseteq \hat{v}$ of that value, we have

$$\hat{p} \sqsubseteq \hat{v}|_{\|\hat{p}\|}$$

*Proof.* By induction on the subvalue relation $\sqsubseteq$.

- Cases $V \sqsubseteq V$ and $\texttt{[]} \sqsubseteq V$ are trivially true.
- Case $\texttt{[}V\texttt{]}^b \sqsubseteq \texttt{[}V'\texttt{]}^b$: We have $\texttt{[}V'\texttt{]}^b|_{\|\texttt{[}V\texttt{]}^b\|} = \texttt{[}V'\texttt{]}^b|_{b \cup \|V\|}$ by definition, and $V'|_{\|V\|} \sqsupseteq V$ by the induction hypothesis, and can therefore conclude $\texttt{[}V\texttt{]}^b \sqsubseteq \texttt{[}V'\texttt{]}^b|_{\|\texttt{[}V\texttt{]}^b\|}$.
- The cases for list concatenation and records are similar.

□

$$
\begin{array}{rcl}
\mathfrak{W}[\![O]\!] & = & O \\
\mathfrak{W}[\![A \;\texttt{->}\; B]\!] & = & \mathfrak{W}[\![A]\!] \;\texttt{->}\; \mathfrak{W}[\![B]\!] \\
\mathfrak{W}[\![(l_i : A_i)_{i=1}^{n}]\!] & = & (l_i : \mathfrak{W}[\![A_i]\!])_{i=1}^{n} \\
\mathfrak{W}[\![[A]]\!] & = & [\mathfrak{W}[\![A]\!]] \\
\mathfrak{W}[\![\textbf{Prov}(A)]\!] & = & (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\textbf{String}, \textbf{String}, \textbf{Int})) \\
\mathfrak{W}[\![\textbf{table}(R)]\!] & = & (\textbf{table}(\lfloor R \rfloor), () \;\texttt{->}\; [\mathfrak{W}[\![(R)]\!]])
\end{array}
$$

Figure 17: Type translation for $\mathsf{Links}^{\mathsf{W}}$

**Theorem 6** (Correctness of lineage)**.** *Let $q$ be a monotonic query with $\|q\| = \emptyset$ and let $\hat{\Sigma}$ be a context, such that $q$ evaluates to $\hat{v}$ in $\hat{\Sigma}$: $\hat{\Sigma}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}, \hat{v}$. Then for every sublist $\hat{p} \sqsubseteq \hat{v}$ we can evaluate $q$ in a restricted context $\hat{\Sigma}|_{\|\hat{p}\|}$ to obtain a value $\hat{v}'$ and $\hat{p}$ will be a sublist of $\hat{v}'$.*

$$\forall \hat{p} \sqsubseteq \hat{v} : \hat{\Sigma}|_{\|\hat{p}\|}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}' \wedge \hat{p} \sqsubseteq \hat{v}'$$

*Proof.* Using Corollary 4 of Theorem 3 we have

$$\hat{\Sigma}|_{\|\hat{p}\|}, q|_{\|\hat{p}\|} \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}|_{\|\hat{p}\|}$$

for any $\hat{p}$ and, because of Lemma 5, $\hat{v}|_{\|\hat{p}\|} \sqsupseteq \hat{p}$ so set

$$\hat{v}' = \hat{v}|_{\|\hat{p}\|}$$

Since $q$ has no annotations on its own, it is not affected by restriction: $q|_{\|\hat{p}\|} = q$ and we can conclude that

$$\hat{\Sigma}|_{\|\hat{p}\|}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}' \wedge \hat{p} \sqsubseteq \hat{v}'$$

□

## 5. Provenance translations

In the previous section, we have presented two extensions of Links: $\mathsf{Links}^{\mathsf{W}}$, which supports where-provenance in queries, and $\mathsf{Links}^{\mathsf{L}}$, which supports lineage in queries. Here, we show that both extensions can be implemented by a type-preserving source-to-source translation to plain Links.

### 5.1. Where-Provenance

We define a type-directed translation from $\mathsf{Links}^{\mathsf{W}}$ to Links based on the semantics presented in the previous section. The syntactic translation of types $\mathfrak{W}[\![-]\!]$ is shown in Figure 17. We write $\mathfrak{W}[\![\Gamma]\!]$ for the obvious extension of the type translation to contexts. The implementation extends the Links parser and type checker, and desugars the $\mathsf{Links}^{\mathsf{W}}$ AST to a Links AST after type checking, reusing the backend mostly unchanged. The expression translation function is also written $\mathfrak{W}[\![-]\!]$ and is shown in Figure 18.

$$
\begin{array}{rcl}
\mathfrak{W}[\![c]\!] & = & c \\
\mathfrak{W}[\![x]\!] & = & x \\
\mathfrak{W}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{W}[\![M_i]\!])_{i=1}^n \\
\mathfrak{W}[\![N.l]\!] & = & \mathfrak{W}[\![N]\!].l \\
\mathfrak{W}[\![\textsf{fun}(x_i|_{i=0}^n)\ \{M\}]\!] & = & \textsf{fun}(x_i|_{i=0}^n)\ \{\mathfrak{W}[\![M]\!]\} \\
\mathfrak{W}[\![M(N_i|_{i=0}^n)]\!] & = & \mathfrak{W}[\![M]\!](\mathfrak{W}[\![N_i]\!]|_{i=0}^n) \\
\mathfrak{W}[\![\textsf{var}\ x = M; N]\!] & = & \textsf{var}\ x = \mathfrak{W}[\![M]\!]; \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\textsf{query}\ \{M\}]\!] & = & \textsf{query}\ \{\mathfrak{W}[\![M]\!]\} \\
\mathfrak{W}[\![[]]\!] & = & [] \\
\mathfrak{W}[\![[M]]\!] & = & [\mathfrak{W}[\![M]\!]] \\
\mathfrak{W}[\![M \texttt{ ++ } N]\!] & = & \mathfrak{W}[\![M]\!] \texttt{ ++ } \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\textsf{if}\ (L)\ \{M\}\ \textsf{else}\ \{N\}]\!] & = & \textsf{if}\ (\mathfrak{W}[\![L]\!])\ \{\mathfrak{W}[\![M]\!]\}\ \textsf{else}\ \{\mathfrak{W}[\![N]\!]\} \\
\mathfrak{W}[\![\textsf{empty}\ (M)]\!] & = & \textsf{empty}\ (\mathfrak{W}[\![M]\!]) \\
\mathfrak{W}[\![\textsf{for}\ (x \texttt{ <- } L)\ M]\!] & = & \textsf{for}\ (x \texttt{ <- } \mathfrak{W}[\![L]\!])\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\textsf{where}(M)\ N]\!] & = & \textsf{where}(\mathfrak{W}[\![M]\!])\ \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\textsf{for}\ (x \texttt{ <-- } L)\ M]\!] & = & \textsf{for}\ (x \texttt{ <- } \mathfrak{W}[\![L]\!].2())\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\textsf{data}\ M]\!] & = & \mathfrak{W}[\![M]\!].\textsf{data} \\
\mathfrak{W}[\![\textsf{prov}\ M]\!] & = & \mathfrak{W}[\![M]\!].\textsf{prov} \\
\mathfrak{W}[\![\textsf{insert}\ L\ \textsf{values}\ M]\!] & = & \textsf{insert}\ \mathfrak{W}[\![L]\!].1\ \textsf{values}\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\textsf{update}\ (x \texttt{ <- } L)\ \textsf{where}\ M\ \textsf{set}\ N]\!] & = & \textsf{update}\ (x \texttt{ <- } \mathfrak{W}[\![L]\!].1)\ \textsf{where}\ \mathfrak{W}[\![M]\!]\ \textsf{set}\ \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\textsf{delete}\ (x \texttt{ <- } L)\ \textsf{where}\ M]\!] & = & \textsf{delete}\ (x \texttt{ <- } \mathfrak{W}[\![L]\!].1)\ \textsf{where}\ \mathfrak{W}[\![M]\!]
\end{array}
$$

$$
\mathfrak{W}[\![\textsf{table}\,n\,\textsf{with}(R)\textsf{where}\ S]\!] = (\textsf{table}\,n\,\textsf{with}\ (R),\ \ \textsf{fun}()\{\textsf{for}(x \texttt{ <-- } \textsf{table}\,n\,\textsf{with}\ (R))[(R \triangleright_x^n S)]\})
$$

$$
\begin{array}{rcl}
\cdot \triangleright_x^n \cdot & = & \cdot \\
(R, l : O) \triangleright_x^n \cdot & = & (R \triangleright_x^n \cdot), l = x.l \\
(R, l : O) \triangleright_x^n (S, l\ \textsf{prov default}) & = & (R \triangleright_x^n S), l = (\textsf{data} = x.l, \textsf{prov} = (n, l_d, x.oid)) \\
(R, l : O) \triangleright_x^n (S, l\ \textsf{prov}\ M) & = & (R \triangleright_x^n S), l = (\textsf{data} = x.l, \textsf{prov} = \mathfrak{W}[\![M]\!](x))
\end{array}
$$

Figure 18: Translation of $\mathsf{Links}^{\mathsf{W}}$ to Links, and auxiliary operation $R \triangleright_x^n S$

Values of type **Prov**$(O)$ are represented at runtime as ordinary Links records with type (data: $O$, prov: (String, String, Int)). Thus, the keywords **data** and **prov** translate to projections to the respective fields.

We translate table declarations to pairs. The first component is a simple table declaration where all columns have their primitive underlying non-provenance type. We will use the underlying table declaration for insert, update, and delete operations. The second component is essentially a delayed query that calculates where-provenance for the entire table. (The fact that it is delayed is important here, because it means that it can be inlined and simplified later, rather than loaded into memory.) We compute provenance for each record by iterating over the table. For every record of the input table, we construct a new record with the same fields as the table. For every column with provenance, the field's value is a record with data and prov fields. The data field is just the value. The translation of table references also uses an auxiliary operation $R \triangleright_x^n S$ which, given a row type $R$, a table name $n$, a variable $x$ and a provenance specification

$S$, constructs a record in which each field contains data from $x$ along with the specified provenance (if any). We wrap the iteration in an anonymous function to delay execution: otherwise, the provenance-annotated table would be constructed in memory when the table reference is first evaluated. We will eventually apply this function in a query, and the Links query normalizer will inline the provenance annotations and normalize them along with the rest of the query.

We translate table comprehensions to comprehensions over the second component of a translated table declaration. Since that component is a function, we have to apply it to a (unit) argument.

For example, recall the example query q1''' from Section 2, Figure 4. The table declaration translates as follows:

```
var agencies = (table "Agencies"
                    with (name: String, based_in: String, phone: String),
 fun () { for (t <-- table "Agencies"
                          with (name: String, based_in: String, phone: String))
   [(name = t.name, based_in = t.based_in,
     phone = (data = t.phone, prov = ("Agencies", "phone", t.oid)))] })
```

The translation of the externalTours table reference is similar, but simpler, since it has no **prov** annotations. The query translates to

```
query {
  for (a <-- agencies.2())
    for (e <-- externalTours.2())
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone.data, p_phone = a.phone.prov)]
}
```

Moreover, after inlining the adjusted definitions of agencies and externalTours and normalizing, the provenance computations in the delayed query agencies.2 are also inlined, resulting in the following SQL query. In this query, the table and column part of the where-provenance are in fact static, and the generated SQL query reflects this by using constants in the **select** clause. We see no trace of function application, or nested record projections in the guise of **data** and **prov**.

```
select
  e.name as name,
  a.phone as phone,
  'agencies' as p_phone_1,
  'phone' as p_phone_2,
  a.oid as p_phone_3
from
  Agencies as a,
  ExternalTours as e
where
  a.name = e.name and e.type = 'boat'
```

The type-preservation correctness property of the where-provenance translation is that it preserves well-formedness. We first need

**Lemma 7.** Let $R$ be a row and $S$ be a provenance specification. Then

- $\mathfrak{W}[\![(\lvert R \rvert)]\!] = (R)$.

- $\lfloor(R \rhd S)\rfloor = (R)$.

The type-preservation property for the translation is stated as follows and proved in Appendix B.3:

**Theorem 8.**

1. *For every* $\mathsf{Links^W}$ *context* $\Gamma$*, term* $M$*, and type* $A$*, if* $\Gamma \vdash_{\mathsf{Links^W}} M : A$ *then* $\mathfrak{W}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![A]\!]$*.*
2. *For every* $\mathsf{Links^W}$ *context* $\Gamma$*, provenance specification* $S$*, row* $R$ *and subrow* $R'$ *such that* $R' \rhd_x^n S$ *is defined, if* $\Gamma \vdash S : \mathsf{ProvSpec}(R)$ *then* $\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash (R' \rhd_x^n S) : \mathfrak{W}[\![(R' \rhd S)]\!]$*.*

We have shown that annotation-propagation in $\mathsf{Links^W}$ is color-propagating (Theorem 2) and that the translation to $\mathsf{Links}$ is type-preserving (Theorem 8). We have not, however, shown that the translation correctly implements the semantics. This is intuitively clear, but a formal proof is nontrivial because a single step in $\mathsf{Links^W}$ can translate to multiple steps in $\mathsf{Links}$, involving terms that have no $\mathsf{Links^W}$ counterpart.

*5.2. Lineage*

We define a typed translation from $\mathsf{Links^L}$ to $\mathsf{Links}$. The translation has two parts: an outer translation called *doubling* ($\mathfrak{D}$) and an inner part called *lineage translation* ($\mathfrak{L}$). The former is used for translating ordinary $\mathsf{Links^L}$ code while the latter is used to translate query code inside a **lineage** keyword. The syntactic translation of $\mathsf{Links^L}$ types for the doubling translation is shown in Figure 12, and the translation used for the lineage translation is the $\mathfrak{L}$ translation shown earlier. We write $\mathfrak{D}[\![\Gamma]\!]$ and $\mathfrak{L}[\![\Gamma]\!]$ for the obvious extensions of these translations to contexts.

The translation of $\mathsf{Links^L}$ expressions to $\mathsf{Links}$ is shown in Figures 20–22. Following the type translation, term translation operates in two modes: $\mathfrak{D}$ and $\mathfrak{L}$. We translate ordinary $\mathsf{Links}$ programs using the translation $\mathfrak{D}[\![-]\!]$. When we reach a **lineage** block, we switch to using the $\mathfrak{L}[\![-]\!]$ translation. $\mathfrak{L}[\![[M]]\!]$ provides initial lineage for list literals. Their lineage is simply empty. Table comprehension is the most interesting case. We translate a table iteration **for** $(x$ <-- $L)$ $M$ to a nested list comprehension. The outer comprehension binds $y$ to the results of the lineage-computing view of $L$. The inner comprehension binds a fresh variable $z$, iterating over $\mathfrak{L}[\![M]\!]$—the original comprehension body $M$ transformed using $\mathfrak{L}$. The original comprehension body $M$ is defined in terms of $x$, which is not bound in the transformed comprehension. We therefore replace every occurrence of $x$ in $\mathfrak{L}[\![e]\!]$ by $y.\mathsf{data}$. In the body of the nested comprehension we thus have $y$, referring to the table row annotated with lineage, and $z$, referring to the result of the original comprehension's body, also annotated with lineage. As the result of our transformed comprehension, we return the plain data part of $z$ as our data, and the combined lineage annotations of $y$ and $z$ as our provenance. (Handling **where**-clauses is straightforward, as shown in Figure 21.)

One subtlety here is that lineage blocks need not be closed, and so may refer to variables that were defined (and will be bound to values at runtime) outside of the lineage block. This could cause problems: for example, if we bind $x$ to a collection $[1, 2, 3]$ outside a lineage block and refer to it in a comprehension inside such a block, then uses of $x$ will expect the collection elements to be

$$
\begin{array}{rcl}
\mathfrak{D}[\![O]\!] & = & O \\
\mathfrak{D}[\![A \texttt{ -> } B]\!] & = & (\mathfrak{D}[\![A]\!] \texttt{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \texttt{ -> } \mathfrak{L}[\![B]\!]) \\
\mathfrak{D}[\![(l_i : A_i)_{i=1}^n]\!] & = & (l_i : \mathfrak{D}[\![A_i]\!])_{i=1}^n \\
\mathfrak{D}[\![[A]]\!] & = & [\mathfrak{D}[\![A]\!]] \\
\mathfrak{D}[\![\textsf{table}(R)]\!] & = & (\textsf{table}(R), () \texttt{ -> } \mathfrak{L}[\![[(R)]]\!])
\end{array}
$$

Figure 19: Doubling translation

records such as $(\mathsf{data} = 1, \mathsf{prov} = L)$ rather than plain numbers. Therefore, such variables need to be adjusted so that they will have appropriate structure to be used within a lineage block. The auxiliary type-indexed function $d2l[\![A]\!]$ in Figure 22 accomplishes this by mapping a value of type $\mathfrak{D}[\![A]\!]$ to one of type $\mathfrak{L}[\![A]\!]$. We define $\mathfrak{L}^*[\![-]\!]$ as a function that applies $\mathfrak{L}[\![-]\!]$ to its argument and substitutes all free variables $x : A$ with $d2l[\![A]\!](x)$.

The $\mathfrak{D}[\![-]\!]$ translation also has to account for functions that are defined outside lineage blocks but may be called either outside or inside a lineage block. To support this, the case for functions in the $\mathfrak{D}[\![-]\!]$ translation creates a pair, whose first component is the recursive $\mathfrak{D}[\![-]\!]$ translation of the function, and whose second component uses the $\mathfrak{L}^*[\![-]\!]$ translation to create a version of the function callable from within a lineage block. (We use $\mathfrak{L}^*[\![-]\!]$ because functions also need not be closed.) Function calls outside lineage blocks are translated to project out the first component; function calls inside such blocks are translated to project out the second component (this is actually accomplished via the $A$ -> $B$ case of $d2l$.)

Finally, notice that the $\mathfrak{D}[\![-]\!]$ translation maps table types and table references to pairs. This is similar to the $\mathfrak{W}[\![-]\!]$ translation, so we do not explain it in further detail; the main difference is that we just use the oid field to assign default provenance to all rows.

For example, if we wrap the query from Figure 1 in a **lineage** block it will be rewritten to this:

```
for (y_a <- agencies.2())
  for (z_a <- for (y_e <- externalTours.2())
                for (z_e <- [(data = (name = y_a.data.name, phone = y_a.data.phone),
                              prov = [])])
                where (y_a.data.name == y_e.data.name && y_e.data.type == "boat")
                  [(data = z_e.data, prov = y_e.prov ++ z_e.prov)])
    [(data = z_a.data, prov = y_a.prov ++ z_a.prov)]
```

Once agencies and externalTours are inlined, Links's built-in normalization algorithm simplifies this query to:

```
for (y_a <- table "Agencies" with ...)
  for (y_e <- table "ExternalTours" with ...)
  where (y_a.data.name == y_e.data.name && y_e.data.type == "boat")
    [(data = (name = y_a.data.name,phone = y_a.data.phone),
      prov = [("Agencies", y_a.oid), ("ExternalTours",y_e.oid)])]
```

Before considering the main type-preservation result, we state some auxiliary lemmas with corresponding proofs in Appendix B.4:

$$
\begin{array}{rcl}
\mathfrak{D}[\![c]\!] & = & c \\
\mathfrak{D}[\![x]\!] & = & x \\
\mathfrak{D}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{D}[\![M_i]\!])_{i=1}^n \\
\mathfrak{D}[\![N.l]\!] & = & \mathfrak{D}[\![N]\!].l \\
\mathfrak{D}[\![\textsf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] & = & (\textsf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{D}[\![M]\!]\}, \mathfrak{L}^*[\![\textsf{fun}(x_i|_{i=1}^n)\ \{M\}]\!]) \\
\mathfrak{D}[\![M(N_i|_{i=1}^n)]\!] & = & \mathfrak{D}[\![M]\!].1(\mathfrak{D}[\![N_i]\!]_{i=1}^n) \\
\mathfrak{D}[\![\textsf{var}\ x = M; N]\!] & = & \textsf{var}\ x = \mathfrak{D}[\![M]\!]; \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![[]]\!] & = & [] \\
\mathfrak{D}[\![[M]]\!] & = & [\mathfrak{D}[\![M]\!]] \\
\mathfrak{D}[\![M \texttt{ ++ } N]\!] & = & \mathfrak{D}[\![M]\!] \texttt{ ++ } \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![\textsf{if}\ (L)\ \{M\}\ \textsf{else}\ \{N\}]\!] & = & \textsf{if}\ (\mathfrak{D}[\![L]\!])\ \{\mathfrak{D}[\![M]\!]\}\ \textsf{else}\ \{\mathfrak{D}[\![N]\!]\} \\
\mathfrak{D}[\![\textsf{query}\ \{M\}]\!] & = & \textsf{query}\ \{\mathfrak{D}[\![M]\!]\} \\
\mathfrak{D}[\![\textsf{empty}\ (M)]\!] & = & \textsf{empty}\ (\mathfrak{D}[\![M]\!]) \\
\mathfrak{D}[\![\textsf{for}\ (x \texttt{ <- } L)\ M]\!] & = & \textsf{for}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!])\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\textsf{where}(M)\ N]\!] & = & \textsf{where}(\mathfrak{D}[\![M]\!])\ \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![\textsf{for}\ (x \texttt{ <-- } L)\ M]\!] & = & \textsf{for}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\textsf{insert}\ L\ \textsf{values}\ M]\!] & = & \textsf{insert}\ \mathfrak{D}[\![L]\!].1\ \textsf{values}\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\textsf{update}\ (x \texttt{ <- } L)\ \textsf{where}\ M\ \textsf{set}\ \mathfrak{D}[\![N]\!] & = & \textsf{update}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \textsf{where}\ \mathfrak{D}[\![M]\!]\ \textsf{set}\ N]\!] \\
\mathfrak{D}[\![\textsf{delete}\ (x \texttt{ <- } L)\ \textsf{where}\ M]\!] & = & \textsf{delete}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \textsf{where}\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\textsf{lineage}\ \{M\}]\!] & = & \textsf{query}\ \{\mathfrak{L}^*[\![M]\!]\}
\end{array}
$$

$$
\mathfrak{D}[\![\textsf{table}\ n\ \textsf{with}\ (R)]\!] \quad = \quad (\textsf{table}\ n\ \textsf{with}\ (R),\ \textsf{fun}()\{\mathfrak{L}[\![\textsf{table}\ n\ \textsf{with}\ (R)]\!]\})
$$

Figure 20: Translation of $\mathsf{Links}^{\mathsf{L}}$ to $\mathsf{Links}$: outer translation

**Lemma 9.** 1. If $A :: \mathsf{QType}$ then $\mathfrak{D}[\![A]\!] = \mathfrak{D}[\![\mathfrak{L}[\![A]\!]]\!]$.
2. If $\Gamma \vdash M : \mathfrak{D}[\![A]\!]$ then $\Gamma \vdash d2l(M) : \mathfrak{L}[\![A]\!]$.

The type-preservation property for the translation from $\mathsf{Links}^{\mathsf{L}}$ to $\mathsf{Links}$ is stated as follows:

**Theorem 10.** *Let $M$ be given such that $\Gamma \vdash_{\mathsf{Links}^{\mathsf{L}}} M : A$. Then:*

1. $\mathfrak{L}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$
2. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![A]\!]$
3. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![A]\!]$

*Proof.* The proof of the first part is by induction on the structure of typing derivations. The interesting cases are for the List, ForList and ForTable cases, where lineage annotations are created or propagated. The detailed derivations are given in Appendix B.5.

For the second part, suppose $\Gamma \vdash M : A$. Then by part 1 we know $\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$. Clearly, for each $x_i : A_i$ in $\Gamma$ we have $\mathfrak{D}[\![\Gamma]\!] \vdash x_i : \mathfrak{D}[\![A_i]\!]$, so it follows that $\mathfrak{D}[\![\Gamma]\!] \vdash d2l(x_i) : \mathfrak{L}[\![A_i]\!]$ for each $i$ by Lemma 9(2). Using the (standard) substitution lemma for $\mathsf{Links}$ typing, we can conclude $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![A]\!]$.

$$
\begin{array}{rcl}
\mathfrak{L}[\![c]\!] & = & c \\
\mathfrak{L}[\![x]\!] & = & x \\
\mathfrak{L}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{L}[\![M_i]\!])_{i=1}^n \\
\mathfrak{L}[\![N.l]\!] & = & \mathfrak{L}[\![N]\!].l \\
\mathfrak{L}[\![\textbf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] & = & (\textbf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{L}[\![M]\!]\}) \\
\mathfrak{L}[\![M(N_i|_{i=1}^n)]\!] & = & \mathfrak{L}[\![M]\!](\mathfrak{L}[\![N_i]\!]|_{i=1}^n) \\
\mathfrak{L}[\![\textbf{var}\ x = M; N]\!] & = & \textbf{var}\ x = \mathfrak{L}[\![M]\!]; \mathfrak{L}[\![N]\!] \\
\mathfrak{L}[\![\mathsf{[]}]\!] & = & \mathsf{[]} \\
\mathfrak{L}[\![[M]]\!] & = & [(\mathsf{data} = \mathfrak{L}[\![M]\!], \mathsf{prov} = \mathsf{[]})] \\
\mathfrak{L}[\![M \text{ ++ } N]\!] & = & \mathfrak{L}[\![M]\!] \text{ ++ } \mathfrak{L}[\![N]\!] \\
\mathfrak{L}[\![\textbf{if}\ (L)\ \{M\}\ \textbf{else}\ \{N\}]\!] & = & \textbf{if}\ (\mathfrak{L}[\![L]\!])\ \{\mathfrak{L}[\![M]\!]\}\ \textbf{else}\ \{\mathfrak{L}[\![N]\!]\} \\
\mathfrak{L}[\![\textbf{query}\ \{M\}]\!] & = & \textbf{query}\ \{\mathfrak{L}[\![M]\!]\} \\
\mathfrak{L}[\![\textbf{empty}\ (M)]\!] & = & \textbf{empty}\ (\mathfrak{L}[\![M]\!]) \\
\mathfrak{L}[\![\textbf{for}\ (x \text{ <- } L)\ M]\!] & = & \quad \textbf{for}\ (y \text{ <- } \mathfrak{L}[\![L]\!]) \\
 & & \qquad \textbf{for}\ (z \text{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) \\
 & & \qquad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \text{ ++ } z.\mathsf{prov})] \\
\mathfrak{L}[\![\textbf{where}(M)\ N]\!] & = & \textbf{where}(\mathfrak{L}[\![M]\!])\ (\mathfrak{L}[\![N]\!]) \\
\mathfrak{L}[\![\textbf{for}\ (x \text{ <-- } L)\ M]\!] & = & \quad \textbf{for}\ (y \text{ <- } \mathfrak{L}[\![L]\!]) \\
 & & \qquad \textbf{for}\ (z \text{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) \\
 & & \qquad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \text{ ++ } z.\mathsf{prov})] \\
\mathfrak{L}[\![\textbf{lineage}\ \{M\}]\!] & = & \textbf{query}\ \{\mathfrak{L}[\![M]\!]\}
\end{array}
$$

$$
\mathfrak{L}[\![\textbf{table}\ n\ \textbf{with}\ (R)]\!] \quad = \quad \textbf{for}(x \text{ <-- } \textbf{table}\ n\ \textbf{with}\ (R))[(\mathsf{data} = x, \mathsf{prov} = [(n, x.\mathsf{oid})])]
$$

Figure 21: Translation of $\mathsf{Links}^{\mathsf{L}}$ to $\mathsf{Links}$: inner translation

Finally, for the third part, again the proof is by induction on the structure of the derivation of $\Gamma \vdash M : A$. Most cases are straightforward; we show a few representative cases for (single-argument) functions and the **lineage** keyword, illustrating the need for duplicating code in the type translation for functions and the use of $\mathfrak{L}^*[\![-]\!]$. The cases for updates and table references are similar to those for $\mathsf{Links}^{\mathsf{W}}$, but simpler because the types of the fields do not change in the translation from $\mathsf{Links}^{\mathsf{L}}$ to $\mathsf{Links}$. We illustrate the case for translation of functions, since it is one of the subtler cases; the cases for function application and the **lineage** keyword are given in the appendix. If the derivation is of the form:

$$
\textsc{Fun}\quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \textbf{fun}\ (x)\{M\} : A \text{ -> } B}
$$

then by induction we have $\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!]$ and by part 2 we know that $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\textbf{fun}\ (x)\{M\}]\!] : \mathfrak{L}[\![(A) \text{ -> } B]\!]$. We can proceed as follows:

$$
\begin{array}{ll}
\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!] & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \textbf{fun}\ (x)\{\mathfrak{D}[\![M]\!]\} : \mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!] & \text{by rule} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\textbf{fun}\ (x)\{M\}]\!] : \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!] & \text{by part 2} \\
\mathfrak{D}[\![\Gamma]\!] \vdash (\textbf{fun}\ (x)\{\mathfrak{D}[\![M]\!]\}, \mathfrak{L}^*[\![\textbf{fun}\ (x)\{M\}]\!]) : \mathfrak{D}[\![A \text{ -> } B]\!] & \text{by rule}
\end{array}
$$

$$
\begin{array}{rcl}
\mathfrak{L}^*[\![M]\!] & = & \mathfrak{L}[\![M]\!][x_i \mapsto d2l[\![A_i]\!](x_i)|_{i=1}^{n}] \\
 & & \text{where } x_1 : A_1, \ldots, x_n : A_n \text{ are the free variables of } M \\
d2l[\![A]\!] & : & \mathfrak{D}[\![A]\!] \texttt{ -> } \mathfrak{L}[\![A]\!] \\
d2l[\![O]\!](x) & = & x \\
d2l[\![A \texttt{ -> } B]\!](f) & = & f.2 \\
d2l[\![(l_1 : A_1, \ldots, l_n : A_n)]\!](x) & = & (l_1 : d2l[\![A_1]\!](x.l_1), \ldots, l_n : d2l[\![A_n]\!](x.l_n)) \\
d2l[\![[A]]\!](y) & = & \textbf{for}(x \texttt{ <- } y)[(\mathsf{data} = d2l[\![A]\!](x), \mathsf{prov} = [])] \\
d2l[\![\textbf{table}(R)]\!](t) & = & t.2()
\end{array}
$$

Figure 22: Translation of Links$^{\mathsf{L}}$ to Links: term translation

where the final step relies on the fact that $\mathfrak{D}[\![A \texttt{ -> } B]\!] = (\mathfrak{D}[\![A]\!] \texttt{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \texttt{ -> } \mathfrak{L}[\![B]\!])$. □

As with the where-provenance translation, we have proven the correctness of lineage annotation propagation (Theorem 6) and type-preservation of the translation (Theorem 10). The latter is a partial sanity check, but no proof, that this translation faithfully implements the semantics.

## 6. Experimental Evaluation

We implemented the two variants of Links with language-integrated provenance, Links$^{\mathsf{W}}$ and Links$^{\mathsf{L}}$, featuring our extensions for where-provenance and lineage, respectively. In this section we compare them against plain Links on a number of queries to determine their overhead. We also compare both variants against *Perm*, a database-integrated provenance system.

Both provenance variants of Links build on its query shredding capabilities as described by Cheney et al. [11]. They used queries against a simple test database schema (see Figure 23) that models an organization with departments, employees and external contacts. We adapt some of their benchmarks to return where-provenance and lineage and compare against the same queries without provenance.

Unlike Cheney et al. [11] our database does not include an additional id field, instead we use PostgreSQL's OIDs, which are used for identification of rows in where-provenance and lineage. We populate the databases at varying sizes using randomly generated data in the same way Cheney et al. [11] describe it: "We vary the number of departments in the organization from 4 to 4096 (by powers of 2). Each department has on average 100 employees and each employee has 0–2 tasks." The largest database, with 4096 departments, is 142 MB on disk when exported by `pg_dump` to a SQL file (excluding OIDs). We create additional indices on tasks(employee), tasks(task), employees(dept), and contacts(dept).

All tests were performed on an otherwise idle desktop system with a 3.2 GHz quad-core CPU, 8 GB RAM, and a 500 GB HDD. The system ran Linux (kernel 4.5.0) and we used PostgreSQL 9.4.2 as the database engine. Links and its variants Links$^{\mathsf{W}}$ and Links$^{\mathsf{L}}$ are interpreters written in OCaml, which were compiled to native code using OCaml 4.02.3. The exact versions of Links$^{\mathsf{W}}$ and Links$^{\mathsf{L}}$ used for this set of benchmarks can be downloaded from `https://www.inf.ed.ac.`

`uk/research/isdd/admin/package?download=188` and `https://www.inf.ed.ac.uk/research/isdd/admin/package?download=189` respectively.

*6.1. Where-provenance*

To be usable in practice, where-provenance should not have unreasonable runtime overhead. We compare queries *without* any where-provenance against queries that calculate where-provenance on *some* of the result and queries that calculate *full* where-provenance wherever possible. This should give us an idea of the overhead of where-provenance on typical queries, which are somewhere in between full and no provenance.

The nature of where-provenance suggests two hypotheses: First, we expect the asymptotic cost of where-provenance-annotated queries to be the same as that of regular queries. Second, since every single piece of data is annotated with a triple, we expect the runtime of a fully where-provenance-annotated query to be at most four times the runtime of an unannotated query just for handling more data.

We only benchmark *default* where-provenance, that is table name, column name, and the database-generated OID for row identification. External provenance is computed by user-defined database-executable functions and can thus be arbitrarily expensive.

We use the queries with nested results from Cheney et al. [11] and use them unchanged for comparison with the two variants with varying amounts of where-provenance.

For *full* where-provenance we change the table declarations to add provenance to every field, except the OID. The full declarations can be found in Figure C.33. This changes the types, so we have to adapt the queries and some of the helper functions used inside the queries, see Figure C.35. Figure 24 shows the benchmark queries with full provenance. See Appendix C for the full code, including table declarations and helper functions. Note that for example query Q2 maps the **data** keyword over the employees tasks before comparing the tasks against *"abstract"*. Query Q6 returns the outliers in terms of salary and their tasks, concatenated with the clients, who are assigned the *fake* task *"buy"*. Since the fake task is not a database value it cannot have where-provenance. $\mathsf{Links}^{\mathsf{W}}$ type system prevents us from pretending it does. Thus, the list of tasks has type [String], not [Prov(String)].

The queries with *some* where-provenance are derived from the queries with full provenance. Query Q1 drops provenance from the contacts' fields. Q2 returns data and provenance separately. It does not actually return less information, it is just less type-safe. Q3 drops provenance from the employee. Q4 returns the employees' provenance only, and drops the actual data. Q5 does not return provenance on the employees fields. Q6 drops provenance on the department. (These queries make use of some auxiliary functions which are included in the appendix.)

**Setup.** We have three $\mathsf{Links}^{\mathsf{W}}$ programs, one for each level of where-provenance annotations. For each database size, we drop all tables and load a dump from disk, starting with 4096. We then run $\mathsf{Links}^{\mathsf{W}}$ three times, once for each program in order *all*, *some*, *none*. Each of the three programs performs and times its queries 5 times in a row and reports the median runtime in milliseconds. The programs measure runtime using the $\mathsf{Links}^{\mathsf{W}}$ built-in function serverTimeMilliseconds which in turn uses OCaml's Unix.gettimeofday.

```
table departments with (oid: Int, name: String)
table employees with (oid: Int, dept: String, name: String, salary: Int)
table tasks with (oid: Int, employee: String, task: String)
table contacts with (oid: Int, dept: String, name: String, client: Bool)
```

Figure 23: Benchmark database schema, cf. Cheney et al. [11].

```
# Q1 : [(contacts: [("client": Prov(Bool), name: Prov(String))], ...
for (d <-- departments)
  [(contacts = contactsOfDept(d),
    employees = employeesOfDept(d),
    name = d.name)]

# Q2 : [(d: Prov(String))]
for (d <- q1())
where (all(d.employees, fun (e) {
      contains(map(fun (x) { data x }, e.tasks), "abstract") }))
  [(d = d.name)]

# Q3 : [(b: [Prov(String)], e: Prov(String))]
for (e <-- employees)
  [(b = tasksOfEmp(e), e = e.name)]

# Q4 : [(dpt:Prov(String), emps:[Prov(String)])]
for (d <-- departments)
  [(dpt = (d.name),
    emps = for (e <-- employees)
            where ((data d.name) == (data e.dept))
              [(e.name)])]

# Q5 : [(a: Prov(String), b: [(name: Prov(String), ...
for (t <-- tasks)
  [(a = t.task, b = employeesByTask(t))]

# Q6 : [(d: Prov(String), p: [(name: Prov(String), tasks: [String])])]
for (x <- q1())
  [(d = x.name,
    p = get(outliers(x.employees),
          fun (y) { map(fun (z) { data z }, y.tasks) }) ++
        get(clients(x.contacts), fun (y) { ["buy"] }))]
```

Figure 24: “allprov” benchmark queries used in experiments

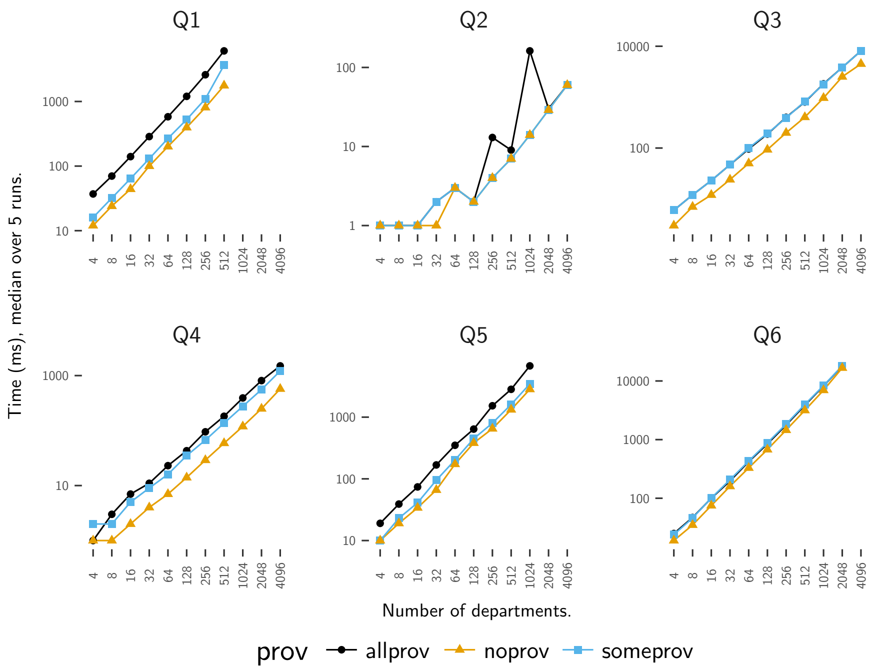


Figure 25: Where-provenance query runtimes.

| Query | median runtime* in ms | | | overall slowdown |
|---|---|---|---|---|
| | allprov | someprov | noprov | (geom mean) |
| Q1 | 6068 | 3653 | 1763 | 2.26 |
| Q2 | 60 | 60 | 60 | 1.52 |
| Q3 | 8100 | 8064 | 4497 | 1.88 |
| Q4 | 1502 | 1214 | 573 | 2.8 |
| Q5 | 6778 | 3457 | 2832 | 1.85 |
| Q6 | 17874 | 18092 | 16716 | 1.22 |

Figure 26: Median runtimes for largest dataset (Q1 at 512 departments, Q5 at 1024 departments, Q6 at 2048 departments, others at 4096 departments) and geometric means of overall slowdowns.

**Data.** Figure 25 shows our experimental results. We have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Note that both axes are logarithmic. Measurements of full where-provenance are in black circles, no provenance are yellow triangles, some provenance is blue squares. Based on test runs we had to exclude some results for queries at larger database sizes because the queries returned results that were too large for Links to construct as in-memory values.

The graph for query Q2 looks a bit odd. This seems to be due to Q2 not actually returning any data for some database sizes, because for some of the (randomly generated) instances there just are no departments where all employees have the task *"abstract"*.

The table in Figure 26 lists all queries with their median runtimes with full, some, and no provenance. The time reported is in milliseconds, for the largest database instance that both variants of a query ran on. For most queries this is 4096; for Q1 it is 512, 1024 for Q5, and 2048 for Q6. Figure 26 also reports the slowdown of full where-provenance versus no provenance as the geometric mean across all database sizes, for each query. The slowdown ranges from 1.22 for query Q6 up to 2.8 for query Q4. Note that query Q2 has the same runtime for all variants at 4096 departments, but full provenance is slower for some database sizes, so the overall slowdown is $> 1$.

**Interpretation.** The graphs suggest that the asymptotic cost of all three variants is the same, confirming our hypothesis. This was expected, anything else would have suggested a bug in our implementation.

The multiplicative overhead seems to be larger for queries that return more data. Notably, for query Q2, which returns no data at all on some of our test database instances, the overhead is hardly visible. The raw amount of data returned for the full where-provenance queries is three to four times that of a plain query. Most strings are short names and provenance adds two short strings and a number for table, column, and row. The largest overhead is 2.8 for query Q4, which exceeds our expectations due to just raw additional data needing to be processed.

### *6.2. Lineage*

We expect lineage to have different performance characteristics than where-provenance. Unlike where-provenance, lineage is conceptually set valued. A query with few actual results could have huge lineage, because lineage is combined for equal data. In practice, due to Links using multiset semantics for queries, the amount of lineage is bounded by the shape of the query. Thus, we expect lineage queries to have the same asymptotic cost as queries without lineage. However, the lineage translation still replaces single comprehensions by nested comprehensions that combine lineage. We expect this to have a larger impact on performance than where-provenance, where we only needed to trace more data through a query.

Figure 27 lists the queries used in the lineage experiments. For lineage, queries are wrapped in a **lineage** block. Our implementation does not currently handle function calls in lineage blocks automatically, so in our experiments we have manually written lineage-enabled versions of the functions employeesByTask and tasksOfEmp, whose bodies are wrapped in a **lineage** block. We reuse some of the queries from the where-provenance experiments, namely Q3, Q4, and Q5.

```
typename Lin(a) = (data: a, prov: [(String, Int)]);

# AQ6 : [Lin((department: String, outliers: [Lin((name: String, ...
for (d <- for (d <-- departments)
          [(employees = for (e <-- employees) where (d.name == e.dept)
                          [(name = e.name, salary = e.salary)],
            name = d.name)])
  [(department = d.name,
    outliers = for (o <- d.employees) where (o.salary > 1000000 || o.salary < 1000) [o])]

# Q3 : [Lin((b: [Lin(String)]), e: String)]
for (e <-- employees) [(b = tasksOfEmp(e), e = e.name)]

# Q4 : [Lin((dpt: String, emps: [Lin(String)]))]
for (d <-- departments)
  [(dpt = d.name,
    emps = for (e <-- employees) where (d.name == e.dept) [e.name])]

# Q5 : [Lin((a: String, b: [Lin((name: String, salary: Int, ...
for (t <-- tasks) [(a = t.task, b = employeesByTask(t))]

# Q6N : [Lin((department: String, people:[Lin((name: String, ...
for (x <-- departments)
  [(department = x.name,
    people = (for (y <-- employees)
               where (x.name == y.dept && (y.salary < 1000 || y.salary > 1000000))
                 [(name = y.name,
                   tasks = for (z <-- tasks) where (z.employee == y.name) [z.task])])
          ++  (for (y <-- contacts) where (x.name == y.dept && y."client")
                 [(name = y.dept, tasks = ["buy"])]))]

# Q7 : [Lin((department: String, employee: (name: String, ...
for (d <-- departments) for (e <-- employees)
where (d.name == e.dept && e.salary > 1000000 || e.salary < 1000)
  [(employee = (name = e.name, salary = e.salary), department = d.name)]

# QC4 : [Lin((a: String, b: String, c: [Lin((doer: String, ...
for (x <-- employees) for (y <-- employees)
where (x.dept == y.dept && x.name <> y.name)
  [(a = x.name, b = y.name,
    c = (for (t <-- tasks) where (x.name == t.employee) [(doer = "a", task = t.task)])
     ++ (for (t <-- tasks) where (y.name == t.employee) [(doer = "b", task = t.task)]))]

# QF3 : [Lin((String, String))]
for (e1 <-- employees) for (e2 <-- employees)
where (e1.dept == e2.dept && e1.salary == e2.salary && e1.name <> e2.name)
  [(e1.name, e2.name)]

# QF4 : [Lin(String)]
(for (t <-- tasks) where (t.task == "abstract") [t.employee]) ++
(for (e <-- employees) where (e.salary > 50000) [e.name])
```

Figure 27: Lineage queries used in experiments

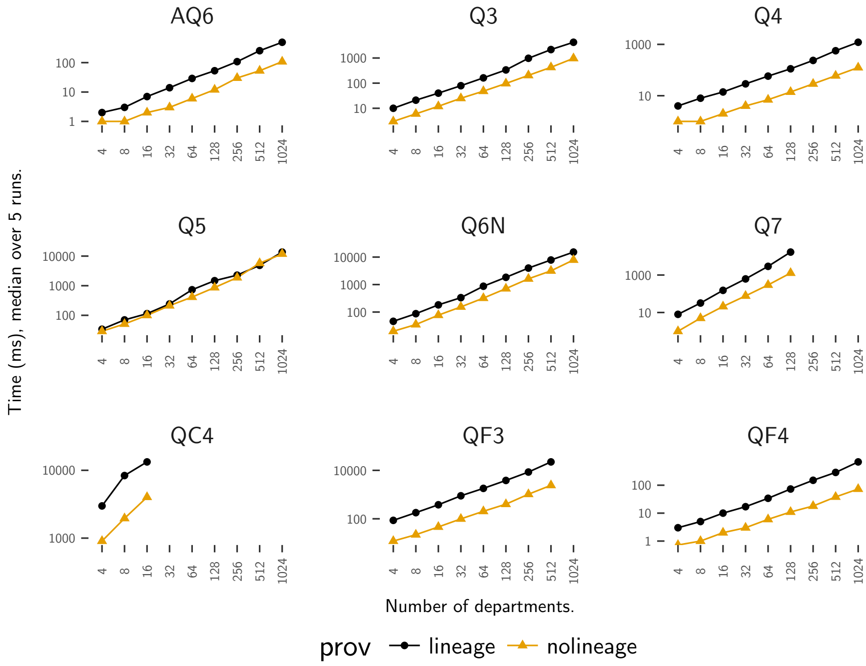


Figure 28: Lineage query runtimes.

| Query | median runtime in ms lineage | nolineage | overall slowdown (geom mean) |
|---|---|---|---|
| AQ6 | 493 | 108 | 3.8 |
| Q3 | 4234 | 969 | 3.76 |
| Q4 | 1208 | 125 | 7.55 |
| Q5 | 13662 | 11851 | 1.25 |
| Q6N | 15200 | 7872 | 2.38 |
| Q7 | 16766 | 1283 | 4.17 |
| QC4 | 13291 | 4021 | 1.53 |
| QF3 | 22298 | 2412 | 6.71 |
| QF4 | 682 | 73 | 6.49 |

Figure 29: Median runtimes at largest dataset (Q7 at 128 departments, QC4 at 16 departments, QF3 at 512 departments, others at 1024 departments) and geometric means of overall slowdowns

Queries AQ6, Q6N, and Q7 are inspired by query Q6, but not quite the same. Queries QF3 and QF4 are two of the flat queries from Cheney et al. [11]. Query QC4 computes pairs of employees in the same department and their tasks in a "tagged union". Again, these queries employ some helper functions which are included in an appendix.

We use a similar experimental setup to the one for where-provenance. We only use databases up to 1024 departments, because most of the queries are a lot more expensive. Query QC4 has excessive runtime even for very small databases. Query Q7 ran out of memory for larger databases. We excluded them from runs on larger databases.

**Data.** Figure 28 shows our lineage experiment results. Again, we have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Both axes are logarithmic. Measurements with lineage are in black circles, no lineage is shown as yellow triangles.

The table in Figure 29 lists queries and their median runtimes with and without lineage. The time reported is in milliseconds, for the largest database instance that both variants of a query ran on. For most queries this is 1024; for Q7 it is 128, 16 for QC4, and 512 for QF3. The table also reports the slowdown of lineage versus no lineage as the geometric mean over all database sizes. (We exclude database size 4 for the mean slowdown in QF4 which reported taking 0 ms for no lineage queries which would make the geometric mean infinity.) The performance penalty for using lineage ranges from query Q5 needing a quarter more time to query Q4 being more than 7 times slower than its counterpart.

**Interpretation.** Due to Links multiset semantics, we do not expect lineage to cause an asymptotic complexity increase. The experiments confirm this. Lineage is still somewhat expensive to compute, with slowdowns ranging from 1.25 to more than 7 times slower. Further investigation of the SQL queries generated by shredding is needed.

### *6.3. Threats to validity*

Our test databases are only moderately sized. However, our result sets are relatively large. Query Q1 for example returns the whole database in a different shape. Links' runtime representation of values in general and database results in particular has a large memory overhead. In practice, for large databases we should avoid holding the whole result in memory. This should reduce the overhead (in terms of memory) of provenance significantly. (It is not entirely clear how to do this in the presence of nested results and thus query shredding.) In general, it looks like the overhead of provenance is dependent on the amount of data returned. It would be good to investigate this more thoroughly. Also, it could be advantageous to represent provenance in a special way. In theory, we could store the relation and column name in a more compact way, for example.

One of the envisioned main use cases of provenance is debugging. Typically, a user would filter a query anyway to pin down a problem and thus only look at a small number of results and thus also query less provenance. Our experiments do not measure this scenario but instead compute provenance for all query results eagerly. Thus, the slowdown factors we showed represent worst case upper bounds that may not be experienced in common usage patterns.

Our measurements do not include program rewriting time. However, this time is only dependent on the lexical size of the program and is thus fairly small and, most importantly, independent of the database size. Since Links is interpreted, it does not really make sense to distinguish translation time from execution time, but both the where-provenance translation and the lineage translation could happen at compile time, leaving only slightly larger expressions to be normalized at runtime. Across the queries above, the largest observed time spent rewriting $\mathsf{Links}^{\mathsf{W}}$ or $\mathsf{Links}^{\mathsf{L}}$ to plain Links was 5 milliseconds with the arithmetic mean coming to 0.5 milliseconds.

### *6.4. Comparison with Perm*

In this section we compare $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ to *Perm* [23], as an instance of a database-integrated provenance system. This is very much a comparison between apples and oranges.

The subset of queries supported by both Links variants and *Perm* is limited. Most of the queries above use nested results which are not supported by *Perm*. Many common flat relational queries use aggregations which are not supported by Links. Others do not have large or interesting provenance annotations, be it where-provenance or lineage.

For this comparison we use a synthetic dataset. We create tables of integers $1, \ldots, n$ for $n = (10000, 100000, 1000000)$; a simple string representation of the number; an English language cardinal like "one", "two", ...; and an English language ordinal ("first", "second", ...).

| i | s | cardinal | ordinal |
|---|---|---|---|
| 1 | "1" | "one" | "first" |
| 2 | "2" | "two" | "second" |
| ⋮ | | | |
| $n$ | "$n$" | "$en$" | "$n$th" |

We create 64 copies of these tables at each size $n$ and call them i_s_c_o_$n$_1, i_s_c_o_$n$_2, .... Their content is the same, but their OIDs are distinct. The data loading scripts are 55 MB, 640 MB, and 7.8 GB on disk.

We use the same machine as before to run both databases and database clients. We use *Perm* version 0.1.1, which is a fork of Postgres 8.3 which adds support for provenance. We compiled from source, which required passing `-fno-aggressive-loop-optimizations` to GCC 6.3.1 as it would otherwise miscompile. This seems to be a known problem with Postgres 8.3, which *Perm* 0.1.1 is based on. Links uses the current version of Postgres as its database backend, which is Postgres 9.6.3.

In this set of benchmarks, we measure wall clock time of single runs. Links queries execute the query and print the result to stdout which is ignored. Printing uses Links's native format with pretty printing (line breaks and indentation) disabled. *Perm* queries are executed using psql with a "harness" like this:

```
\COPY (SQL query goes here) TO STDOUT WITH CSV
```

#### *6.4.1. Where-provenance*

We use a family of queries that join $m = (16, 32, 64)$ of the tables described above on their integer column and select the provenance-annotated cardinal column for each of them. Thus, the where-provenance $\mathsf{Links}^{\mathsf{W}}$ queries look like this (table declarations are in Appendix C.1):

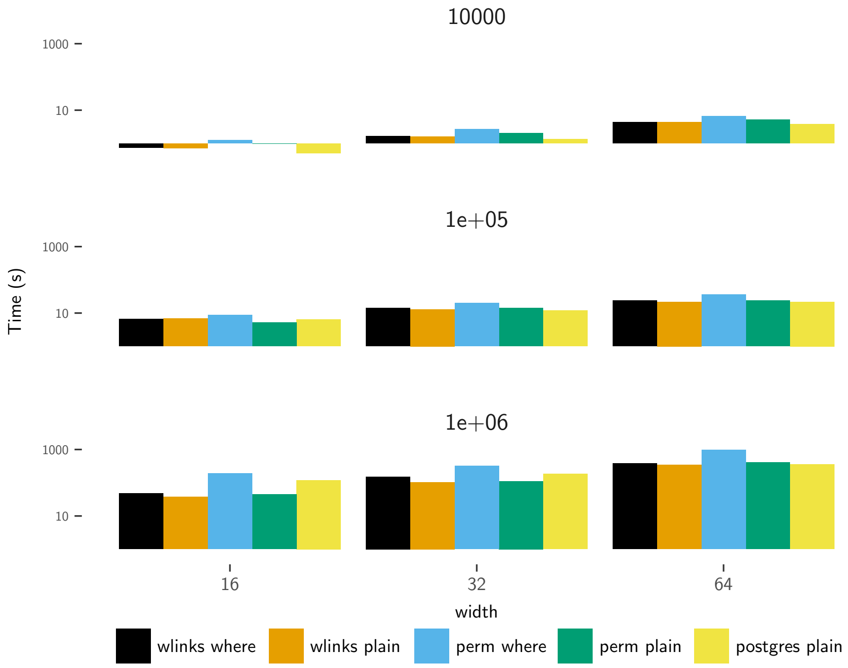


Figure 30: Where-provenance times grouped by table size ($n$) and number of tables ($m$). Note that wlinks and postgres queries are filtered, perm queries are not.

```
query {
  for (t_1 <-- i_s_c_o_n_1) ... for (t_m <-- i_s_c_o_n_m)
  where (mod(t_1.i, 100) < 5 && t_1.i == t_2.i && ... && t_1.i = t_m.i)
    [(c1 = t_1.cardinal, c2 = t_2.cardinal, ..., cm = t_m.cardinal)] }
```

Testing revealed that Links$^{W}$ runs out of memory for the largest (n=1000000,m=64) query. Rather than using smaller input databases, we filtered the result using `mod(t_1.i, 100) < 5` as an additional condition in the where clause.

Unfortunately, *Perm's* where-provenance support is too restrictive and refuses to execute an equivalent query with the following error message: "WHERE-CS only supports conjunctive equality comparisons in WHERE clause." Fortunately, *Perm* has no problems computing the full result, so we used queries of the following form, *without* filtering based on `t_1.i % 100 < 5`.

```
SELECT PROVENANCE ON CONTRIBUTION (WHERE)
    t_1.cardinal AS c1, ..., t_m.cardinal AS cm
FROM i_s_c_o_n_1 AS t_1, ..., i_s_c_o_n_m AS t_m
WHERE t_1.i = t_2.i AND ... AND t_1.i = t_m.i
```

We execute variants without where-provenance of both the Links$^{W}$ and *Perm* queries. For Links$^{W}$ we keep the table declarations as they are, but use the **data** keyword to project to just the data and rely on query normalization to not compute provenance. We run a fifth set of queries against Postgres 9.6.3 which are just like the plain *Perm* queries, but *with* filtering, like the Links$^{W}$ queries.

Figure 30 shows query runtimes in seconds grouped by size of tables ($n$) and number of tables joined ($m$). Keep in mind that the *Perm* variants return a lot more data. In the table below we show result size in megabytes at $n = 1000000$ for $\mathsf{Links}^{\mathsf{W}}$ with where-provenance annotations, *Perm* with annotations, and Postgres without annotations. We measure the size simply as byte count of the printed result. Examples of the output can be found in Appendix C.1.

| | m=16 | m=32 | m=64 |
|---|---|---|---|
| $\mathsf{Links}^{\mathsf{W}}$ | 89.2 MB | 179.1 MB | 359.1 MB |
| *Perm* | 1589.3 MB | 3187.5 MB | 6384.0 MB |
| Postgres | 37.2 MB | 74.3 MB | 148.6 MB |

Looking at the runtime difference between the *Perm* queries without where-provenance and the plain Postgres queries we see that the result size does not have a great impact on runtime. In general, the numbers between systems are hard to compare, not just because of result size. We only consider one family of highly synthetic queries and the experimental setup is not necessarily a realistic reflection of any real-world use. However, we do observe some trends: The runtime difference between processing 10x data (going down one row in the graph) is larger than the difference between systems, by far. Doubling the number of tables considered also dominates difference between systems. We conclude that the overhead of where-provenance in both *Perm* and $\mathsf{Links}^{\mathsf{W}}$ is moderate and the systems are roughly comparable.

*6.4.2. Lineage*

We use the same data as before and similar queries to compare $\mathsf{Links}^{\mathsf{L}}$ to *Perm Influence Contribution Semantics* (PI-CS). Lineage and PI-CS are not equivalent in general [21], but for the queries we use here the annotations contain, more or less, the same information.

We use a family of queries similar to those for where-provenance. Again we join $m = (16, 32, 64)$ tables, but this time we return only the first table's integer and English cardinal columns, and their lineage. The number of joins is particularly interesting here because it increases the size of the provenance metadata without affecting the actual result size.

We run variants with lineage and PI-CS metadata, as well as just the plain queries. Finally, we run the plain version of the *Perm* query against the Postgres database used by $\mathsf{Links}^{\mathsf{L}}$. This time all variants, including *Perm*, are filtered to 5% of the result size, as seen below. The $\mathsf{Links}^{\mathsf{L}}$ query and example output can be found in Appendix C.1.

```
SELECT PROVENANCE t_1.i, t_1.cardinal
FROM i_s_c_o_n_1 AS t_1, ..., i_s_c_o_n_m AS t_m
WHERE t_1.i % 100 < 5 AND t_1.i = t_2.i AND ... AND t_(m − 1).i = t_m.i
```

Instead of a list of annotations per result row, *Perm* produces wider tables, adding columns to identify join partners. Table rows are identified by the whole width, so for $m = 64$ joined tables we have two columns for the actual result and $64 * 4$ columns of provenance metadata. The example result below is transposed.

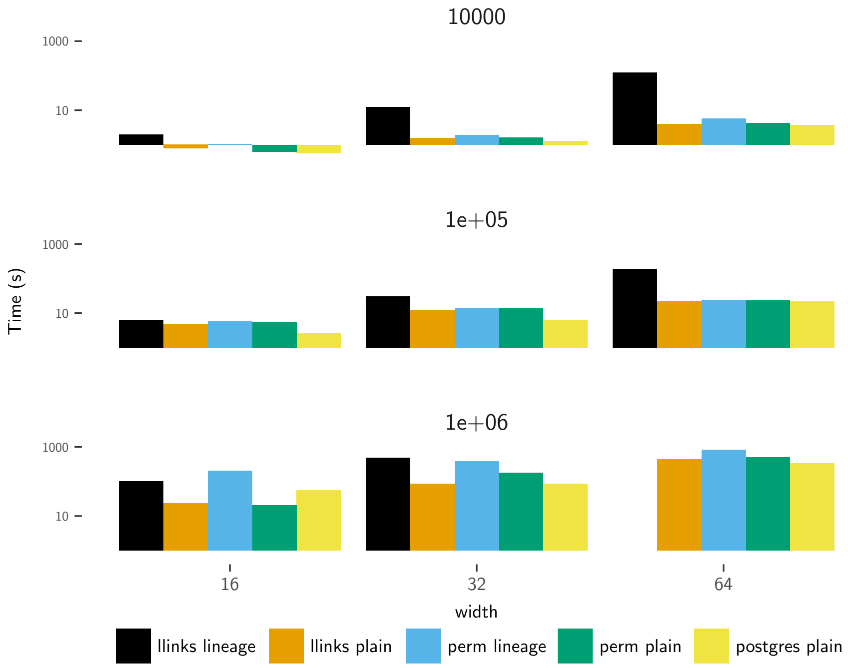


Figure 31: Lineage times grouped by relation size ($n$) and width ($m$). All queries are filtered to return only 5% of results.

| | | | |
|---|---|---|---|
| i | 1 | 2 | ... |
| cardinal | one | two | ... |
| prov_public_i_s_c_o_1000_1_i | 1 | 2 | ... |
| prov_public_i_s_c_o_1000_1_s | 1 | 2 | ... |
| prov_public_i_s_c_o_1000_1_cardinal | one | two | ... |
| prov_public_i_s_c_o_1000_1_ordinal | first | second | ... |
| ⋮ | | | |

We show query runtimes grouped by size of the tables ($n$) and number of tables joined ($m$) in Figure 31. We omitted the largest Links$^{\mathsf{L}}$ query (n=1000000, m=64); it ran for 33745 seconds, which would have distorted the graph too much. This query just barely did not run out of memory, causing severe GC thrashing and leaving little memory for the database server and disk caches.

These timings are whole program execution and so include pre- and post-processing steps. Links$^{\mathsf{L}}$ is translated to plain Links, as described in Section 5.2, which took less than 1 millisecond for all queries. Query normalization for the lineage queries takes around 9 milliseconds for m=16, 41 milliseconds for m=32, and 194 milliseconds for m=64. Postprocessing times (with data already in memory) range from almost 10 seconds for the lineage query at n=1000000, m=64 to 11 milliseconds for n=10000, m=16.

The queries executed by Postgres are on average a bit faster than the same queries executed by *Perm*. We did not investigate this further, a simple explanation would be that Postgres 9.6.3 is just a bit faster than Postgres 8.3 which is the version *Perm* was forked from.

Below we show result size at $n = 1000000$ for plain queries, and lineage queries at $m = 16$ and $m = 32$. We measure the size simply as byte count of the printed result. In some ways, the data is a worst case for *Perm*, because the width of the result is so much smaller than the width of the annotations. We can see this clearly in the result size table above. Despite that, the query execution time overhead of lineage annotations is remarkably low in *Perm*.

| system | plain | lineage (m=16) | lineage (m=32) |
|---|---|---|---|
| $\mathsf{Links}^{\mathsf{L}}$ | 3.1 MB | 38.5 MB | 73.7 MB |
| *Perm* | 2.7 MB | 89.4 MB | 176.2 MB |

*Perm* considerably outperforms $\mathsf{Links}^{\mathsf{L}}$ when it comes to lineage computation. Their performance on plain queries is similar, which comes at a bit of a surprise. We expected $\mathsf{Links}^{\mathsf{L}}$ to be a worse database client than the native `psql` client, even for flat queries. This can partly be explained by experiment setup. We had database clients and servers run on the same machine to avoid network issues. However, this reduces the amount of memory available for caching, especially since $\mathsf{Links}^{\mathsf{L}}$ uses so much memory to nearly run out on some queries. This means a lot of time is spent by the database system waiting for disk seeks and postprocessing time is low by comparison. Except for the largest queries, postprocessing by $\mathsf{Links}^{\mathsf{L}}$ is typically well below 1 second.

We take away three things: (1) A different experimental setup could alleviate memory pressure and cache behavior and bring out processing times. (2) We could change Links to emit queries that use *Perm's* built-in provenance features when possible. (3) Most interesting would be to look at different ways to rewrite $\mathsf{Links}^{\mathsf{L}}$ queries. Currently, we use Links's nested query capabilities which allow a fairly naive translation. *Perm* exploits the fact that lineage is bounded by the structure of the query, adding columns instead of nested data. Perhaps we could do something similar in $\mathsf{Links}^{\mathsf{L}}$.

## 7. Related Work

Buneman et al. [5] gave the first definition of where-provenance in the context of a semistructured data model. The DBNotes system of Bhagwat et al. [3] supported where-provenance via SQL query extensions. DBNotes provides several kinds of where-provenance in conjunctive SQL queries, implemented by translating SQL queries to one or more provenance-propagating queries. Buneman et al. [6] proposed a where-provenance model for nested relational calculus queries and updates, and proved expressiveness results. They observed that where-provenance could be implemented by translating and normalizing queries but did not implement this idea; our approach to where-provenance in $\mathsf{Links}^{\mathsf{W}}$ is directly inspired by that idea and is (to the best of our knowledge) the first implementation of it. One important difference is that we *explicitly* manage where-provenance via the **Prov** type, and allow the programmer to decide whether to track provenance for some, all or no fields. Our approach also allows inspecting and comparing the provenance annotations, which Buneman et al. [6] did not allow; nevertheless, our type system prevents the programmer from forging or unintentionally discarding provenance. On the other hand, our approach requires manual data and prov annotations because it distinguishes between raw data and provenance-annotated data.

$\mathsf{Links}^{\mathsf{L}}$ is inspired by prior work on lineage [16] and why-provenance [5]. There have been several implementations of lineage and why-provenance. Cui and Widom implemented lineage in a prototype data warehousing system called WHIPS. The Trio system of Benjelloun et al. [2] also supported lineage and used it for evaluating probabilistic queries; lineage was implemented by defining customized versions of database operations via user-defined functions, which are difficult for database systems to optimize. Glavic and Alonso [23] introduced the Perm system, which translated ordinary queries to queries that compute their own lineage; they handled a larger sublanguage of SQL than previous systems such as Trio, and subsequently Glavic and Alonso [22] extended this approach to handle queries with nested subqueries (e.g. SQL's EXISTS, ALL or ANY operations). They implemented these rewriting algorithms inside the database system and showed performance improvements of up to 30 times relative to Trio. In another line of work, Corcoran et al. [15] and Swamy et al. [37] developed SELinks, a variant of Links with sophisticated support for security policies, including a form of provenance tracking implemented using database extensions and type-based coercions. Our approach instead shows that it is feasible to perform this rewriting outside the database system and leverage the standard SQL interface and underlying query optimization of relational databases.

Both $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ rely on the conservativity and query normalization results that underlie Links's implementation of language-integrated query, particularly Cooper's work (2009) extending conservativity to queries involving higher-order functions, and previous work by Cheney et al. [11] on "query shredding", that is, evaluating queries with nested results efficiently by translation to equivalent flat queries. There are alternative solutions to this problem that support larger subsets of SQL, such as grouping and aggregation, which are not currently supported by Links. There are other approaches to nested data or grouping and aggregation, such as Grust et al.'s *loop-lifting* ([27]) and more recent work on *query flattening* [39] in the Database Supported Haskell (DSH) library, or Suzuki et al.'s QueΛ [36], and it would be interesting to evaluate the performance of these techniques on provenance queries, or to extend Links's query support to grouping and aggregation.

Other authors, starting with Green et al. [25], have proposed provenance models based on annotations drawn from algebraic structures such as semirings. While initially restricted to conjunctive queries, the semiring provenance model has subsequently been extended to handle negation and aggregation operations [1]. Karvounarakis et al. [28] developed ProQL, an implementation of the semiring model in a relational database via SQL query extensions. Glavic et al. [24] present further details of the Perm approach described above, show that semiring provenance can be extracted from Perm's provenance model, and also describe a row-level form of where-provenance. It is not yet clear how to support other instances of the semiring model via query rewriting in Links.

$\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ are currently separate extensions, and cannot be used simultaneously, so another natural area for investigation is supporting multiple provenance models at the same time. We are currently investigating this; one possible difficulty may be the need to combine multiple type translations. We intend to explore this further (and consider alternative models). Cheney et al. [9] presented a general form of provenance for nested relational calculus based on execution traces, and showed how such traces can be used to provide "slices" that explain specific results. While this model appears to generalize all of the aforementioned approaches, it appears nontrivial to implement by translation to relational queries, because it is not obvious how to represent the traces in this approach in a relational data model. (Giorgidze et al. [20] show how to support *nonrecursive* algebraic data types in queries, but the trace datatype is recursive.) This would be a challenging area for future work.

Our translation for lineage is similar in some respects to the doubling translation used in Cheney et al. [10] to compile a simplified form of Links to a F#-like core language. Both translations introduce space overhead and overhead for normal function calls due to pair projections. Developing a more efficient alternative translation (perhaps in combination with a more efficient and more complete compilation strategy) is an interesting topic for future work.

As in most work on provenance, we have focused on explaining questionable results in terms of the source data, and we assume that the query itself is correct and not the source of the problem. It would also be interesting to consider a different problem where the query (or other parts of the program) might have errors, and the question is to identify which parts of the query or program could have contributed to erroneous data. This would require a combination of program slicing [32] and query slicing [9] techniques.

## 8. Conclusions

This article makes several contributions regarding integrating provenance management with programming languages. First, we present language extensions to the Links web programming language that accommodate where-provenance ($\mathsf{Links}^{\mathsf{W}}$) and lineage ($\mathsf{Links}^{\mathsf{L}}$), give their semantics, and establish basic provenance correctness properties. Second, we show how to implement both extensions by translation back to plain Links, relying on Links's existing sophisticated support for language-integrated query, normalization and nested queries.

Our approach shows that it is feasible to implement provenance by rewriting queries *outside* the database system, so that a standard database management system can be used. By building on the well-developed theory of query normalization that underlies Links's approach to language-integrated query, our translations remain relatively simple, while still being translated to SQL queries that are executed efficiently on the database. To the best of our knowledge, our approach is the first efficient implementation of provenance for *nested* query results or for queries that can employ *first-class functions*; at any rate, SQL does not provide either feature. Our results show that provenance for database queries can be implemented efficiently and safely at the language-level. This is a promising first step towards systematic programming language support for provenance.

Links is a research prototype language, but the underlying ideas of our approach could be applied to other systems that support comprehension-based language-integrated query, such as F# and Database Supported Haskell. There are a number of possible next steps, including extending Links's language-integrated query capabilities to support richer queries and more forms of provenance. Another area for future work is establishing the correctness of the provenance translations. We believe it would be better to develop a general translation that abstracts the two given in this article, and prove its correctness once and for all. Finally, we have placed some restrictions on the correctness properties for $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$: specifically, we have not considered the impact of updates on provenance correctness, and we have restricted attention to monotonic queries for $\mathsf{Links}^{\mathsf{L}}$. Lifting restrictions in a satisfying way is also an intriguing direction for future work.

*Acknowledgments.* We would like to James McKinna and several anonymous reviewers for comments and helpful suggestions on this work. This work was supported by EU FP7 project DIACHRON (grant no. 601043) and by a Google Research Award.

[1] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS 2011*, pages 153–164, 2011.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[3] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

[4] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1): 3–48, 1995.

[5] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, number 1973 in LNCS, pages 316–330. Springer, 2001.

[6] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4):28:1–28:47, Dec. 2008.

[7] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, Apr. 2009.

[8] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21:1301–1337, 12 2011. doi: 10.1017/S0960129511000211.

[9] J. Cheney, A. Ahmed, and U. A. Acar. Database queries that explain their work. In *PPDP 2014*, pages 271–282. ACM, 2014.

[10] J. Cheney, S. Lindley, G. Radanne, and P. Wadler. Effective quotation: Relating approaches to language-integrated query. In *PEPM 2014*, pages 15–26. ACM, 2014.

[11] J. Cheney, S. Lindley, and P. Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *SIGMOD 2014*, pages 1027–1038. ACM, 2014.

[12] A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL 2015*, pages 153–165. ACM, 2015.

[13] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL 2009*, volume 5708 of *LNCS*, pages 36–51. Springer, 2009.

[14] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO 2006*, pages 266–296. Springer-Verlag, 2007.

[15] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD 2009*, pages 269–282, 2009.

[16] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[17] S. Fehrenbach and J. Cheney. Language-integrated provenance. In *PPDP 2016*, pages 214–227. ACM, 2016.

[18] M. Felleisen and R. Hieb. The revised report on the syntactic theories of control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[19] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the ferry: Database-supported program execution for Haskell. In *IFL 2010*, pages 1–18. Springer-Verlag, 2011.

[20] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers. Algebraic data types for language-integrated queries. In *DDFP 2013*, pages 5–10. ACM, 2013.

[21] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010. URL `http://cs.iit.edu/%7edbgroup/pdfpubls/G10a.pdf`.

[22] B. Glavic and G. Alonso. Provenance for nested subqueries. In *EDBT 2009*, pages 982–993, 2009.

[23] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE 2009*, pages 174–185, 2009.

[24] B. Glavic, R. Miller, and G. Alonso. Using SQL for efficient generation and querying of provenance information. In *Festschrift in Honour of Peter Buneman*, volume 8000 of *LNCS*, pages 291–320. Springer, 2013.

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

[26] T. Grust and A. Ulrich. First-class functions for first-order database engines. In *DBPL 2013*, 2013.

[27] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1):162–172, 2010.

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

[29] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI 2012*, pages 91–102. ACM, 2012.

[30] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD 2006*, pages 706–706. ACM, 2006.

[31] A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP 2011*, pages 307–319. ACM, 2011.

[32] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP*, pages 365–376. ACM, 2012.

[33] P. Pialorsi and M. Russo. *Introducing Microsoft® LINQ*. Microsoft Press, 2007.

[34] M. Serrano. Hop, a fast server for the diffuse web. In *COORDINATION*, 2009.

[35] L. K. Shar and H. B. K. Tan. Defeating SQL injection. *IEEE Computer*, 46(3):69–77, 2013.

[36] K. Suzuki, O. Kiselyov, and Y. Kameyama. Finally, safely-extensible and efficient language-integrated query. In *PEPM 2016*, pages 37–48, 2016.

[37] N. Swamy, M. W. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *ICFP 2009*, pages 329–340, 2009.

[38] D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML Workshop*, 2006.

[39] A. Ulrich and T. Grust. The flatter, the better: Query compilation based on the flattening transformation. In *SIGMOD 2015*, pages 1421–1426. ACM, 2015.

[40] Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB 1990*, pages 519–538, 1990.

[41] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.

## Appendix A. Notation

| Notation | Sec. | Meaning |
|---|---|---|
| $\Sigma, M \longrightarrow \Sigma', M'$ | 3 | Database state $\Sigma$ and expression $M$ evaluate in one step to $\Sigma'$ and $M'$ |
| $\Sigma, M \longrightarrow^* \Sigma', M'$ | 3 | Reflexive, transitive closure of $\longrightarrow$ |
| $A :: \mathsf{QType}$ | 3 | Type $A$ is allowed as a query result type |
| $R :: \mathsf{BaseRow}$ | 3 | Row $R$ contains only fields of base types |
| $\Gamma \vdash M : A$ | 3 | In type context $\Gamma$, expression $M$ has type $A$ |
| $\Gamma \vdash S : \mathsf{ProvSpec}(R)$ | 4.1 | In type context $\Gamma$, specification $S$ is a valid provenance specification matching $R$ |
| $\lfloor A \rfloor$ | 4.1 | Erasure of $A$, replacing occurrences of $\mathbf{Prov}(O)$ with $O$ |
| $R \triangleright S$ | 4.1 | Augment row $R$ with provenance specification $S$ |
| $cso_\Sigma(M)$ | 4.1 | Set of colored subobjects of expression $M$, with respect to database state $\Sigma$ |
| $\mathfrak{L}[\![A]\!]$ | 4.2 | Lineage type translation of type $A$ |
| $\hat{\Sigma}, M \longrightarrow_{\mathsf{L}} \hat{\Sigma}', M'$ | 4.2 | Lineage-enabled evaluation |
| $\lVert M \rVert$ | 4.2 | Collection of all lineage annotations from $M$ |
| $M\|_b$ | 4.2 | Restriction of $M$ to collection elements whose lineage is contained in $b$ |
| $V \sqsubseteq V'$ | 4.2 | $V$ is obtainable from $V'$ by deleting some list elements |
| $\mathfrak{W}[\![A]\!]$ | 5.1 | Where-provenance type translation |
| $\mathfrak{W}[\![M]\!]$ | 5.1 | Where-provenance expression translation |
| $R \triangleright^n_x S$ | 5.1 | A row expression constructing initial provenance from a row of type $R$ with table name $n$ and variable $x$ according to provenance specification $S$ |
| $\mathfrak{D}[\![A]\!]$ | 5.2 | Doubling translation of type $A$ |
| $\mathfrak{D}[\![M]\!]$ | 5.2 | Doubling translation of expression $M$ |
| $\mathfrak{L}[\![M]\!]$ | 5.2 | Lineage translation of query expression $M$ |
| $\mathfrak{L}^*[\![M]\!]$ | 5.2 | Closing lineage translation of $M$ |
| $d2l[\![A]\!](M)$ | 5.2 | Mapping from doubling translation to lineage translation |

## Appendix B. Proofs

*Appendix B.1. Proof of Theorem 2*

The statement of the theorem was:

$$\Sigma, M \longrightarrow \Sigma, N \Rightarrow cso_\Sigma(N) \subseteq cso_\Sigma(M)$$

where $M$ and $N$ are $\mathsf{Links}^{\mathsf{W}}$ terms, and $\Sigma$ is a context that provides annotated table rows.

*Proof.* The proof is by induction on $\longrightarrow$.

- Case $(\textbf{fun}\, f(x_i)\, M)(V_i) \longrightarrow M[f \coloneqq \textbf{fun}\, f(x_i)\, M, x_i \coloneqq V_i]$:

$$\begin{aligned}
cso_\Sigma(M[f \coloneqq \textbf{fun}\, f(x_i)\, M, x_i \coloneqq V_i]) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(\textbf{fun}\, f(x_i)\, M) \cup \bigcup_{i=0}^{n} cso_\Sigma(V_i) \\
&= cso_\Sigma(\textbf{fun}\, f(x_i)\, M) \cup \bigcup_{i=0}^{n} cso_\Sigma(V_i) \\
&= cso_\Sigma\,((\textbf{fun}\, f(x_i)\, M)(V_i))
\end{aligned}$$

- Case $\textbf{var}\, x = V; M \longrightarrow M[x \coloneqq V]$:

$$cso_\Sigma(M[x \coloneqq V]) \subseteq cso_\Sigma(M) \cup cso_\Sigma(V) = cso_\Sigma(\textbf{var}\, x = V; M)$$

- Case $(l_i = V_i)_{i=1}^n.l_k \longrightarrow V_k$ where $1 \leq k \leq n$:

$$\begin{aligned}
cso_\Sigma(V_k) &\subseteq \bigcup_{i=1}^{n} cso_\Sigma(V_i) \\
&= cso_\Sigma((l_i = V_i)_{i=1}^n) \\
&= cso_\Sigma((l_i = V_i)_{i=1}^n.l_k)
\end{aligned}$$

- Case $\textbf{if}\,(\textbf{true})\, M\, \textbf{else}\, N \longrightarrow M$:

$$\begin{aligned}
cso_\Sigma(M) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(N) \\
&= cso_\Sigma(\textbf{if}\,(\textbf{true})\, M\, \textbf{else}\, N)
\end{aligned}$$

- Case $\textbf{if}\,(\textbf{false})\, M\, \textbf{else}\, N \longrightarrow N$:

$$\begin{aligned}
cso_\Sigma(N) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(N) \\
&= cso_\Sigma(\textbf{if}\,(\textbf{false})\, M\, \textbf{else}\, N)
\end{aligned}$$

- Case $\textbf{query}\, M \longrightarrow M$: $cso_\Sigma(M) = cso_\Sigma(\textbf{query}\, M)$.
- Case $\textbf{table}\, n \longrightarrow \Sigma(n)$: $cso_\Sigma(\Sigma(n) = cso_\Sigma(\textbf{table}\, n)$.
- Case $\textbf{empty}(\texttt{[]}) \longrightarrow \textbf{true}$:

$$cso_\Sigma(\textbf{true}) = \emptyset = cso_\Sigma(\textbf{empty}(\texttt{[]}))$$

- Case $\textbf{empty}(V) \longrightarrow \textbf{false}$, where $V \neq \texttt{[]}$:

$$cso_\Sigma(\textbf{false}) = \emptyset \subseteq cso_\Sigma(V) = cso_\Sigma(\textbf{empty}(V))$$

- Case $\textbf{for}\,(x \texttt{<-} \texttt{[]})\, M \longrightarrow \texttt{[]}$:

$$cso_\Sigma(\texttt{[]}) = \emptyset \subseteq cso_\Sigma(\textbf{for}\,(x \texttt{<-} \texttt{[]})\, M)$$

- Case $\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} \texttt{[}V\texttt{]})\,M \longrightarrow M[x \coloneqq V]$:

$$\begin{aligned} cso_\Sigma(M[x \coloneqq V]) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(V) \\ &= cso_\Sigma(\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} \texttt{[}V\texttt{]})\,M) \end{aligned}$$

- Case $\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V \mathbin{+\!\!+} W)\,M \longrightarrow (\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V)\,M) \mathbin{+\!\!+} (\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} W)\,M)$:

$$\begin{aligned} cso_\Sigma(\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V \mathbin{+\!\!+} W)\,M) &= cso_\Sigma(V \mathbin{+\!\!+} W) \cup cso_\Sigma(M) \\ &= cso_\Sigma(V) \cup cso_\Sigma(W) \cup cso_\Sigma(M) \\ &= cso_\Sigma((\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V)\,M) \mathbin{+\!\!+} (\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} W)\,M)) \end{aligned}$$

- Case $\textsf{\textbf{for}}\,(x \mathrel{\texttt{<--}} V)\,M \longrightarrow \textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V)\,M$:

$$\begin{aligned} cso_\Sigma(\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} V)\,M) &= cso_\Sigma(V) \cup cso_\Sigma(M) \\ &= cso_\Sigma(\textsf{\textbf{for}}\,(x \mathrel{\texttt{<--}} V)\,M) \end{aligned}$$

- Case $M \longrightarrow M' \Rightarrow \mathcal{E}[M] \longrightarrow \mathcal{E}[M']$ (evaluation step inside a context):

$$\begin{aligned} cso_\Sigma(\mathcal{E}[M']) &= cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M') && \text{Lemma 1} \\ &\subseteq cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M) && \text{IH} \\ &= cso_\Sigma(\mathcal{E}[M]) && \text{Lemma 1} \end{aligned}$$

□

*Appendix B.2. Full definitions of auxiliary functions for lineage annotation extraction and restriction*

The interesting cases can be found in Figure 16.

We extend $\|\cdot\|$, the lineage annotation collection function, by recursively collecting annotations.

$$\begin{aligned} \|\texttt{[}M\texttt{]}^a\| &= a \cup \|M\| \\ \|\texttt{[]}\| &= \emptyset \\ \|M \mathbin{+\!\!+} N\| &= \|M\| \cup \|N\| \\ \|M^{\cup b}\| &= b \cup \|M\| \\ \|\textsf{\textbf{table}}\,t\| &= \|\hat{\Sigma}(t)\| \\ \|\textsf{\textbf{var}}\,x = M; N\| &= \|M\| \cup \|N\| \\ \|c\| &= c \\ \|(l_i = M_i)_{i=1}^n\| &= \bigcup_{i=1}^n \|M_i\| \\ \|M.l\| &= \|M\| \\ \|\textsf{\textbf{fun}}\,f(x_i|_{i=1}^n)\,M\| &= \textsf{\textbf{fun}}\,f(x_i|_{i=1}^n)\,\|M\| \\ \|\textsf{\textbf{if}}\,(L)\,M\,\textsf{\textbf{else}}\,N\| &= \|L\| \cup \|M\| \cup \|N\| \\ \|\textsf{\textbf{query}}\,M\| &= \|M\| \\ \|\textsf{\textbf{for}}\,(x \mathrel{\texttt{<-}} M)\,N\| &= \|M\| \cup \|N\| \\ \|\textsf{\textbf{for}}\,(x \mathrel{\texttt{<--}} M)\,N\| &= \|M\| \cup \|N\| \end{aligned}$$

We extend $\cdot|_b$, the erasure function, by recursively erasing.

$$
\begin{aligned}
\texttt{[}M\texttt{]}^a|_b &= \begin{cases} \texttt{[}M|_b\texttt{]}^a & \text{if } a \subseteq b \\ \texttt{[]} & \text{otherwise} \end{cases} \\
\texttt{[]}|_b &= \texttt{[]} \\
(M +\!\!+ N)|_b &= M|_b +\!\!+ N|_b \\
M^{\cup a}|_b &= \begin{cases} (M|_b)^{\cup a} & \text{if } a \subseteq b \\ \texttt{[]} & \text{otherwise} \end{cases} \\
\textsf{table}\, t|_b &= \textsf{table}\, t \\
(\textsf{var}\, x = M; N)|_b &= \textsf{var}\, x = M|_b; N|_b \\
c|_b &= c \\
(l_i = M_i)_{i=1}^n|_b &= (l_i = M_i|_b)_{i=1}^n \\
M.l|_b &= (M|_b).l \\
(\textsf{fun}\, f(x_i|_{i=1}^n)\, M)|_b &= \textsf{fun}\, f(x_i|_{i=1}^n)\, (M|_b) \\
(\textsf{if}\, (L)\, M\, \textsf{else}\, N)|_b &= \textsf{if}\, (L|_b)\, M|_b\, \textsf{else}\, N|_b \\
(\textsf{query}\, M)|_b &= \textsf{query}\, (M|_b) \\
(\textsf{for}\, (x \texttt{<-} M)\, N)|_b &= \textsf{for}\, (x \texttt{<-} M|_b)\, N|_b \\
(\textsf{for}\, (x \texttt{<--} M)\, N)|_b &= \textsf{for}\, (x \texttt{<--} M|_b)\, N|_b
\end{aligned}
$$

*Appendix B.3. Proof of Theorem 8*

Recall the statement of the theorem:

1. For every $\mathsf{Links}^{\mathsf{W}}$ context $\Gamma$, term $M$, and type $A$, if $\Gamma \vdash_{\mathsf{Links}^{\mathsf{W}}} M : A$ then $\mathfrak{W}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![A]\!]$.
2. For every $\mathsf{Links}^{\mathsf{W}}$ context $\Gamma$, provenance specification $S$, row $R$ and subrow $R'$ such that $R' \triangleright_x^n S$ is defined, if $\Gamma \vdash S : \mathsf{ProvSpec}(R)$ then $\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash (R' \triangleright_x^n S) : \mathfrak{W}[\![(R' \triangleright S)]\!]$.

*Proof.* Proof is by induction on the structure of $\mathsf{Links}^{\mathsf{W}}$ derivations. Most cases for the first part are immediate; we show some representative examples.

- If the derivation is of the form:

$$\textsc{Data}\quad \frac{\Gamma \vdash M : \textsf{Prov}(A)}{\Gamma \vdash \textsf{data}\ M : A}$$

  then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![\textsf{Prov}(A)]\!]$, and can conclude:

$$\frac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\textsf{String}, \textsf{String}, \textsf{Int}))}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!].\mathsf{data} : \mathfrak{W}[\![A]\!]}$$

- If the derivation is of the form:

$$\textsc{Data}\quad \frac{\Gamma \vdash M : \textsf{Prov}(A)}{\Gamma \vdash \textsf{prov}\ M : (\textsf{String}, \textsf{String}, \textsf{Int})}$$

then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![\mathbf{Prov}(A)]\!]$, and can conclude:

$$\frac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\mathbf{String}, \mathbf{String}, \mathbf{Int}))}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!].\mathsf{prov} : (\mathbf{String}, \mathbf{String}, \mathbf{Int})}$$

- If the derivation is of the form:

$$\textsc{Table}\ \frac{R :: \mathsf{BaseRow} \qquad \Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash \mathbf{table}\ n\ \mathbf{with}\ (R)\ \mathbf{where}\ S : \mathbf{table}(R \triangleright S)}$$

Then since $\|R \triangleright S\| = R$ (Lemma 7) we can conclude:

$$\frac{}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{table}\ n\ \mathbf{with}\ (R) : \mathbf{table}(\|R \triangleright S\|)}$$

and by the second induction hypothesis,

$$\frac{\dfrac{\dfrac{R :: \mathsf{BaseRow}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{table}\ n\ \mathbf{with}\ (R) : \mathbf{table}(R)} \quad \dfrac{\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash (R \triangleright_x^n S) : \mathfrak{W}[\![(R \triangleright S)]\!]}{\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash [(R \triangleright_x^n S)] : [\mathfrak{W}[\![(R \triangleright S)]\!]]}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{for}(x\ \texttt{<--}\ \mathbf{table}\ n\ \mathbf{with}\ (R))[(R \triangleright_x^n S)] : [\mathfrak{W}[\![(R \triangleright S)]\!]]}}{\mathfrak{W}[\![\Gamma]\!] \vdash\ \mathbf{fun}()\{\mathbf{for}(x\ \texttt{<--}\ \mathbf{table}\ n\ \mathbf{with}\ (R))[(R \triangleright_x^n S)]\} : ()\ \texttt{->}\ [\mathfrak{W}[\![(R \triangleright S)]\!]]}$$

- If the derivation is of the form

$$\textsc{For-Table}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (R) \vdash M : [B]}{\Gamma \vdash \mathbf{for}\ (x\ \ \texttt{<--}\ \ L)\ M : [B]}$$

then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : (\mathbf{table}(\|R\|), ()\ \texttt{->}\ [\mathfrak{W}[\![(R)]\!]])$, so we can proceed as follows:

$$\frac{\dfrac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].2 : ()\ \texttt{->}\ [\mathfrak{W}[\![(R)]\!]]}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].2() : [\mathfrak{W}[\![(R)]\!]]} \qquad \mathfrak{W}[\![\Gamma]\!], x : \mathfrak{W}[\![(R)]\!] \vdash \mathfrak{W}[\![M]\!] : [\mathfrak{W}[\![B]\!]]}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{for}\ (x\ \ \texttt{<-}\ \ \mathfrak{W}[\![L]\!].2())\ \mathfrak{W}[\![M]\!] : [\mathfrak{W}[\![B]\!]]}$$

- If the derivation is of the form:

$$\textsc{Delete}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (\|R\|) \vdash M : \mathbf{Bool}}{\Gamma \vdash \mathbf{delete}\ (x\ \texttt{<--}\ L)\ \mathbf{where}\ M : ()}$$

then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : \mathfrak{W}[\![\mathbf{table}(R)]\!]$ and $\mathfrak{W}[\![\Gamma]\!], x : \mathfrak{W}[\![(\|R\|)]\!] \vdash \mathfrak{W}[\![M]\!] : \mathbf{Bool}$.

$$\frac{\dfrac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : (\mathbf{table}(\|R\|), ()\ \texttt{->}\ [(R)])}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].1 : \mathbf{table}(\|R\|)} \qquad \mathfrak{W}[\![\Gamma]\!], x : (\|R\|) \vdash \mathfrak{W}[\![M]\!] : \mathbf{Bool}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{delete}\ (x\ \texttt{<--}\ \mathfrak{W}[\![L]\!].1)\ \mathbf{where}\ \mathfrak{W}[\![M]\!] : ()}$$

For the second part, we proceed by induction on the structure of the derivation of $\Gamma \vdash S : \mathsf{ProvSpec}(R)$. We show one representative case, for derivations of the form

$$\frac{\Gamma \vdash S : \mathsf{ProvSpec}(R) \qquad \Gamma \vdash M : (R)\ \texttt{->}\ (\mathbf{String}, \mathbf{String}, \mathbf{Int})}{\Gamma \vdash S, l\ \mathbf{prov}\ M : \mathsf{ProvSpec}(R)}$$

In this case, by induction we have that $\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash (R' \triangleright^n_x S) : \mathfrak{W}[\![(R' \triangleright S)]\!]$ holds for any subrow $R'$ of $R$, and by the first induction hypothesis we also know that $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![(R)]\!]$ `->` $(\mathsf{String}, \mathsf{String}, \mathsf{Int})$.

Suppose $R', l : O \triangleright^n_x S, l$ **prov** $M$. Then we can conclude that $\mathfrak{W}[\![\Gamma]\!], x{:}(R) \vdash (R', l : \mathsf{Prov}(O) \triangleright^n_x S, l \text{ } \mathbf{prov} \text{ } M) : \mathfrak{W}[\![(R', l : O \triangleright S, l \text{ } \mathbf{prov} \text{ } O)]\!]$ because $(R', l : O \triangleright^n_x S, l \text{ } \mathbf{prov} \text{ } M) = (R' \triangleright^n_x S), l = (\mathbf{data} = x.l, \mathbf{prov} = \mathfrak{W}[\![M]\!](x))$ and $R', l : O \triangleright S, l \text{ } \mathbf{prov} \text{ } O = (R' \triangleright S), l : \mathsf{Prov}(O)$. □

*Appendix B.4. Proof of Lemma 9*

Recall the statement of the lemma:

1. If $A :: \mathsf{QType}$ then $\mathfrak{D}[\![A]\!] = \mathfrak{D}[\![\mathfrak{L}[\![A]\!]]\!]$.
2. If $\Gamma \vdash M : \mathfrak{D}[\![A]\!]$ then $\Gamma \vdash d2l(M) : \mathfrak{L}[\![A]\!]$.

*Proof.* For part 1, the proof is by induction on the derivation of $A :: \mathsf{QType}$, and is straightforward since both $\mathfrak{D}$ and $\mathfrak{L}$ are the identity on types formed only from base types, records or collection types.

For the second part, the proof is by induction on the structure of $A$ but each case is straightforward. We show the interesting cases for function types and collection types:

- If $A = B_1$ `->` $B_2$ then we proceed as follows:

$$\frac{\Gamma \vdash M : (\mathfrak{D}[\![B_1]\!] \texttt{ -> } \mathfrak{D}[\![B_2]\!], \mathfrak{L}[\![B_1]\!] \texttt{ -> } \mathfrak{L}[\![B_2]\!])}{\Gamma \vdash M.2 : \mathfrak{L}[\![B_1]\!] \texttt{ -> } \mathfrak{L}[\![B_2]\!]}$$

  which suffices since $\mathfrak{L}[\![B_1 \texttt{ -> } B_2]\!] = \mathfrak{L}[\![B_1]\!] \texttt{ -> } \mathfrak{L}[\![B_2]\!]$.

- If $A = [B]$ then we proceed as follows:

| | |
|---|---|
| $\Gamma \vdash M : [\mathfrak{D}[\![B]\!]]$ | assumption |
| $\Gamma, x : \mathfrak{D}[\![B]\!] \vdash x : \mathfrak{D}[\![B]\!]$ | by rule |
| $\Gamma, x : \mathfrak{D}[\![B]\!] \vdash d2l[\![B]\!](x) : \mathfrak{L}[\![B]\!]$ | by IH |
| $\Gamma, x : \mathfrak{D}[\![B]\!] \vdash [] : [(\mathsf{String}, \mathsf{Int})]$ | by rule |
| $\Gamma, x : \mathfrak{D}[\![B]\!] \vdash (\mathbf{data} = d2l[\![B]\!](x), \mathbf{prov} = []) : \mathsf{Lin}(\mathfrak{L}[\![B]\!])$ | by rule |
| $\Gamma, x : \mathfrak{D}[\![B]\!] \vdash [(\mathbf{data} = d2l[\![B]\!](x), \mathbf{prov} = [])] : [\mathsf{Lin}(\mathfrak{L}[\![B]\!])]$ | by rule |
| $\Gamma \vdash \mathbf{for} \ (x \texttt{ <- } M) \ [(\mathbf{data} = d2l[\![B]\!](x), \mathbf{prov} = [])] : [\mathsf{Lin}(\mathfrak{L}[\![B]\!])]$ | by rule |

□

*Appendix B.5. Proof of Theorem 10*

Recall the statement of the theorem:

1. $\mathfrak{L}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$
2. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![A]\!]$
3. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathsf{Links}} \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![A]\!]$

*Proof.* For the first part, we show the details of the cases for singleton lists and list comprehensions. Table comprehensions are similar.

- If the derivation is of the form:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : [A]}\ \textsc{List}$$

then we proceed as follows:

$$\begin{array}{ll}
\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!] & \text{by IH} \\
\mathfrak{L}[\![\Gamma]\!] \vdash [] : [(\textbf{String}, \textbf{Int})] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!] \vdash (\textbf{data} = \mathfrak{L}[\![M]\!], \textbf{prov} = []) : \textbf{Lin}(\mathfrak{L}[\![A]\!]) & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!] \vdash [(\textbf{data} = \mathfrak{L}[\![M]\!], \textbf{prov} = [])] : [\textbf{Lin}(\mathfrak{L}[\![A]\!])] & \text{by rule}
\end{array}$$

which suffices since $\mathfrak{L}[\![[A]]\!] = [\textbf{Lin}\mathfrak{L}[\![A]\!]] = [(\textbf{data} : \mathfrak{L}[\![A]\!], \textbf{prov} : [(\textbf{String}, \textbf{Int})])]$.

- If the derivation is of the form:

$$\frac{\Gamma \vdash L : [A] \qquad \Gamma, x : A \vdash M : [B]}{\Gamma \vdash \textbf{for}\ (x \texttt{ <- } L)\ M : [B]}\ \textsc{For-List}$$

then we proceed as follows:

$$\begin{array}{ll}
\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![L]\!] : [\textbf{Lin}(\mathfrak{L}[\![A]\!])] & \text{by IH} \\
\mathfrak{L}[\![\Gamma]\!], x : \mathfrak{L}[\![A]\!] \vdash \mathfrak{L}[\![M]\!] : [\textbf{Lin}(\mathfrak{L}[\![B]\!])] & \text{by IH} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash y.\textbf{data} : \mathfrak{L}[\![A]\!] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}] : \textbf{Lin}(\mathfrak{L}[\![A]\!]) & \text{by substitution} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash z.\mathsf{data} : \mathfrak{L}[\![B]\!] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash y.\mathsf{prov} : [(\textbf{String}, \textbf{Int})] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash z.\mathsf{prov} : [(\textbf{String}, \textbf{Int})] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash y.\mathsf{prov} \texttt{ ++ } z.\mathsf{prov} : [(\textbf{String}, \textbf{Int})] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash & \\
\quad (\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \texttt{ ++ } z.\mathsf{prov}) : \textbf{Lin}(\mathfrak{L}[\![B]\!]) & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]), z : \textbf{Lin}(\mathfrak{L}[\![B]\!]) \vdash & \\
\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \texttt{ ++ } z.\mathsf{prov})] : [\textbf{Lin}(\mathfrak{L}[\![B]\!])] & \text{by rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \textbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash & \\
\quad \textbf{for}\ (z \texttt{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) & \text{by rule} \\
\qquad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \texttt{ ++ } z.\mathsf{prov})] : [\textbf{Lin}(\mathfrak{L}[\![B]\!])] & \\
\mathfrak{L}[\![\Gamma]\!] \vdash \ \textbf{for}\ (y \texttt{ <- } \mathfrak{L}[\![L]\!]) & \text{by rule} \\
\qquad \textbf{for}\ (z \texttt{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) & \\
\qquad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \texttt{ ++ } z.\mathsf{prov})] : [\textbf{Lin}(\mathfrak{L}[\![B]\!])] &
\end{array}$$

Finally, for the third part, we show the interesting cases for functions, function calls, and **lineage**.

- If the derivation is of the form:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \textbf{fun}\ (x)\{M\} : A \texttt{ -> } B}\ \textsc{Fun}$$

then by induction we have $\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!]$ and by part 2 we know that $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\textbf{fun}\ (x)\{M\}]\!] : \mathfrak{L}[\![(A) \texttt{ -> } B]\!]$. We can proceed as follows:

$$\begin{array}{ll}
\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!] & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \textsf{fun}\ (x)\{\mathfrak{D}[\![M]\!]\} : \mathfrak{D}[\![A]\!] \ \texttt{->}\ \mathfrak{D}[\![B]\!] & \text{by rule} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\textsf{fun}\ (x)\{M\}]\!] : \mathfrak{L}[\![A]\!] \ \texttt{->}\ \mathfrak{L}[\![B]\!] & \text{by part 2} \\
\mathfrak{D}[\![\Gamma]\!] \vdash (\textsf{fun}\ (x)\{\mathfrak{D}[\![M]\!]\}, \mathfrak{L}^*[\![\textsf{fun}\ (x)\{M\}]\!]) : \mathfrak{D}[\![A \ \texttt{->}\ B]\!] & \text{by rule}
\end{array}$$

where the final step relies on the fact that $\mathfrak{D}[\![A \ \texttt{->}\ B]\!] = (\mathfrak{D}[\![A]\!] \ \texttt{->}\ \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \ \texttt{->}\ \mathfrak{L}[\![B]\!])$.

- If the derivation is of the form:

$$\frac{\Gamma \vdash M : A \ \texttt{->}\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B}\ \textsc{App}$$

then we proceed as follows:

$$\begin{array}{ll}
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!] : (\mathfrak{D}[\![A]\!] \ \texttt{->}\ \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \ \texttt{->}\ \mathfrak{L}[\![B]\!]) & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!].1 : \mathfrak{D}[\![A]\!] \ \texttt{->}\ \mathfrak{D}[\![B]\!] & \text{by rule} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![N]\!] : \mathfrak{D}[\![A]\!] & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!].1(\mathfrak{D}[\![N]\!]) : \mathfrak{D}[\![B]\!] & \text{by rule}
\end{array}$$

where in the first step we use the fact that $\mathfrak{D}[\![A \ \texttt{->}\ B]\!] = (\mathfrak{D}[\![A]\!] \ \texttt{->}\ \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \ \texttt{->}\ \mathfrak{L}[\![B]\!])$.

- If the derivation is of the form:

$$\frac{\Gamma \vdash M : [A] \qquad A :: \textsf{QType}}{\Gamma \vdash \textsf{lineage}\ \{M\} : \mathfrak{L}[\![[A]]\!]}\ \textsc{Lineage}$$

then by part (2) we know that $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![[A]]\!]$, so we proceed as follows:

$$\frac{\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![M]\!] : [\mathfrak{L}[\![A]\!]] \qquad \mathfrak{L}[\![A]\!] :: \textsf{QType}}{\mathfrak{D}[\![\Gamma]\!] \vdash \textsf{query}\ \{\mathfrak{L}^*[\![M]\!]\} : [\mathfrak{L}[\![A]\!]]}$$

which suffices since $\mathfrak{D}[\![\mathfrak{L}[\![A]\!]]\!] = \mathfrak{D}[\![A]\!]$ by Lemma 9(1).

□

## Appendix C. Benchmark code

This appendix contains the full listings for the where-provenance and lineage benchmarks. Figures C.32 and C.33 show the plain table declarations and declarations with where-provenance, respectively. These tables also include readonly and tablekeys annotations which were suppressed in the main body of the article; the former indicates that a field is read-only and the latter lists the subsets of the fields that uniquely determine the others.

Figure C.34 shows the helper functions used by the plain versions of the queries, and Figure C.35 shows the variants of these functions adapted to work with where-provenance. Some of the functions, such as any, need no changes at all because they are polymorphic. Figure C.36 shows the versions of the queries with some provenance (the someprov benchmarks).

Figures C.37 and C.38 show the plain queries without lineage annotations; these also employ abbreviations from Figure C.34.

```
var db = database "links";

var departments =
  table "departments"
  with (oid: Int, name: String)
  where oid readonly
  tablekeys [["name"],["oid"]]
  from db;

var employees =
  table "employees"
  with (oid: Int, dept: String, name: String, salary : Int)
  where oid readonly
  tablekeys [["name"],["oid"]]
  from db;

var tasks =
  table "tasks"
  with (oid: Int, employee: String, task: String)
  where oid readonly
  tablekeys [["oid"]]
  from db;

var contacts =
  table "contacts"
  with (oid: Int, dept: String, name: String, "client": Bool)
  where oid readonly
  tablekeys [["name"], ["oid"]]
  from db;
```

Figure C.32: Table declarations for lineage, nolin, and noprov queries.

```
var departments =
  table "departments"
  with (oid: Int, name: String)
  where oid readonly, name prov default
  tablekeys [["name"]]
  from db;

var employees =
  table "employees"
  with (oid: Int, dept: String, name: String, salary : Int)
  where oid readonly, dept prov default,
        name prov default, salary prov default
  tablekeys [["name"]]
  from db;

var tasks =
  table "tasks"
  with (oid: Int, employee: String, task: String)
  where oid readonly, employee prov default, task prov default
  tablekeys [["oid"]]
  from db;

var contacts =
  table "contacts"
  with (oid: Int, dept: String, name: String, "client": Bool)
  where oid readonly, dept prov default,
        name prov default, "client" prov default
  tablekeys [["name"]]
  from db;
```

Figure C.33: Table declarations for where-provenance queries (except noprov).

```
sig tasksOfEmp: ((name:String|_)) -> [String]
fun tasksOfEmp(e) {
  for (t <-- tasks) where (t.employee == e.name) [t.task]
}

sig contactsOfDept: ((name:String|_)) -> [("client":Bool,name:String)]
fun contactsOfDept(d) {
  for (c <-- contacts)
  where (d.name == c.dept)
    [("client" = c."client", name = c.name)]
}

sig employeesByTask: ((employee:String|_)) -> [(name:String,salary:Int,tasks:[String])]
fun employeesByTask(t) {
  for (e <-- employees)
    for (d <-- departments)
    where (e.name == t.employee && e.dept == d.name)
      [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))]
}

sig employeesOfDept: ((name:String|_)) -> [(name:String,salary:Int,tasks:[String])]
fun employeesOfDept(d) {
  for (e <-- employees)
  where (d.name == e.dept)
    [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))]
}

sig any : ([a],(a) -a-> Bool) -a-> Bool
fun any(xs,p) { not(empty(for (x <- xs) where (p(x)) [()])) }

sig all : ([a],(a) -a-> Bool) -a-> Bool
fun all(xs, p) { not(any(xs, fun (x) { not(p(x)) })) }

sig contains: ([a], a) -> Bool
fun contains(xs, u) { any(xs, fun (x) { x == u }) }

fun isPoor(x) { x.salary < 1000 }
fun isRich(x) { x.salary > 1000000 }

sig get: ([(name:a::Any|b)], ((name:a::Any|b)) -c-> d::Any)
      -c-> [(name:a::Any,tasks:d::Any)]
fun get(xs, f) {
  for (x <- xs)
    [(name = x.name, tasks = f(x))]
}

sig outliers: ([(salary:Int|a)]) -> [(salary:Int|a)]
fun outliers(xs) { filter(fun (x) { isRich(x) || isPoor(x) }, xs) }

sig clients: ([("client":Bool|a)]) -> [("client":Bool|a)]
fun clients(xs) { filter(fun (x) { x."client" }, xs) }
```

Figure C.34: Helper functions noprov.

```
# the original (allprov) Q1
fun q_org() {
  for (d <-- departments)
    [(contacts = contactsOfDept(d),
      employees = employeesOfDept(d),
      name = d.name)]
}

sig tasksOfEmp: ((name:Prov(String)|_)) -> [Prov(String)]
fun tasksOfEmp(e) {
  for (t <-- tasks)
  where ((data t.employee) == data e.name)
    [t.task]
}

sig contactsOfDept: ((name:Prov(String)|_)) -> [("client":Prov(Bool),name:Prov(String))]
fun contactsOfDept(d) {
  for (c <-- contacts)
  where ((data d.name) == data c.dept)
    [("client" = c."client", name = c.name)]
}

sig employeesByTask: ((employee:Prov(String)|_))
      -> [(name:Prov(String),salary:Prov(Int),tasks:[Prov(String)])]
fun employeesByTask(t) {
  for (e <-- employees)
    for (d <-- departments)
    where ((data e.name) == (data t.employee)
          && (data e.dept) == (data d.name))
      [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))]
}

sig employeesOfDept: ((name:Prov(String)|_))
     -> [(name:Prov(String),salary:Prov(Int),tasks:[Prov(String)])]
fun employeesOfDept(d) {
  for (e <-- employees)
  where ((data d.name) == data e.dept)
    [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))]
}

fun get(xs, f) {
  for (x <- xs) [(name = x.name, tasks = f(x))]
}

sig outliers: ([(salary:Prov(Int)|a)]) -> [(salary:Prov(Int)|a)]
fun outliers(xs) { filter(fun (x) { isRich(x) || isPoor(x) }, xs) }

sig clients: ([("client":Prov(Bool)|a)]) -> [("client":Prov(Bool)|a)]
fun clients(xs) { filter(fun (x) { data x."client" }, xs) }
```

Figure C.35: Helper functions allprov, someprov (use **data** in some places).

```
# Q1
sig q1 : () -> [(contacts: [("client": Bool, name: String)],
                 employees: [(name: Prov(String), salary: Prov(Int),
                              tasks: [Prov(String)])],
                 name: Prov(String))]
fun q1() {  for (d <-- departments)
              [(contacts = for (c <- contactsOfDept(d))
                             [("client" = data c."client", name = data c.name)],
                employees = employeesOfDept(d),
                name = d.name)] }

# Q2
sig q2 : () -> [(d: String, p: (String, String, Int))]
fun q2() { for (d <- q_org())
             where (all(d.employees, fun (e) {
                   contains(map(fun (x) { data x }, e.tasks), "abstract")
                   }))
             [(d = data d.name, p = prov d.name)] }

# Q3: employees with lists of tasks
sig q3 : () -> [(b: [Prov(String)], e: Prov(String))]
fun q3() { for (e <-- employees) [(b = tasksOfEmp(e), e = (e.name))] }

# Q4: departments with lists of employees
sig q4 : () -> [(dpt:Prov(String), emps:[(String, String, Int)])]
fun q4() { for (d <-- departments)
            [(dpt = d.name, emps = for (e <-- employees)
                                     where ((data d.name) == (data e.dept))
                                     [prov e.name])] }

# Q5: Tasks with employees and departments
fun dropProv(l) { map(fun (x) { data x }, l) }

sig q5: () -> [(a: Prov(String), b: [(name: String, salary: Int, tasks: [String])])]
fun q5() { for (t <-- tasks)
             [(a = t.task, b = for (x <- employeesByTask(t))
                                 [(name = data x.name,
                                   salary = data x.salary,
                                   tasks = dropProv(x.tasks))])]
}

# Q6 Drop prov on department.
sig q6: () -> [(department: String, people: [(name: Prov(String), tasks: [String])])]
fun q6() {  for (x <- q_org())
           [(department = data x.name,
             people = get(outliers(x.employees),
                        fun (y) { map(fun (z) { data z }, y.tasks) }) ++
                     get(clients(x.contacts),
                        fun (y) { ["buy"] }))] }
```

Figure C.36: Queries someprov.

```
# AQ6 : [(department: String, outliers: [(name: String, ...
for (d <- for (d <-- departments)
          [(employees = for (e <-- employees)
                         where (d.name == e.dept)
                         [(name = e.name, salary = e.salary)],
            name = d.name)])
  [(department = d.name, outliers = for (o <- d.employees)
                                     where (o.salary > 1000000 || o.salary < 1000)
                                     [o])]

# Q3 : [(b: [String]), e: String)]
for (e <-- employees) [(b = tasksOfEmp(e), e = e.name)]

# Q4 : [(dpt: String, emps: [String]))]
for (d <-- departments)
  [(dpt = d.name, emps = for (e <-- employees)
                          where (d.name == e.dept)
                          [(e.name)])]

# Q5 : [(a: String, b: [(name: String, salary: Int, ...
for (t <-- tasks) [(a = t.task, b = employeesByTask(t))]

# Q6N : [(department: String, people:[(name: String, ...
for (x <-- departments)
  [(department = x.name,
    people = (for (y <-- employees)
               where (x.name == y.dept && (y.salary < 1000 || y.salary > 1000000))
               [(name = y.name,  tasks = for (z <-- tasks)
                                          where (z.employee == y.name)
                                          [z.task])]) ++
              (for (y <-- contacts)
               where (x.name == y.dept && y."client")
               [(name = y.dept, tasks = ["buy"])]))]
```

Figure C.37: Nolineage queries, part 1

```
# Q7 : [(department: String, employee: (name: String, ...
for (d <-- departments)
for (e <-- employees)
where (d.name == e.dept && e.salary > 1000000 || e.salary < 1000)
  [(employee = (name = e.name, salary = e.salary), department = d.name)]

# QC4 : [(a: String, b: String, c: [(doer: String, ...
for (x <-- employees)
for (y <-- employees)
where (x.dept == y.dept && x.name <> y.name)
  [(a = x.name, b = y.name,
    c = (for (t <-- tasks)
         where (x.name == t.employee)
           [(doer = "a", task = t.task)]) ++
          (for (t <-- tasks)
           where (y.name == t.employee)
             [(doer = "b", task = t.task)]))]

# QF3 : [(String, String)]
for (e1 <-- employees)
for (e2 <-- employees)
where (e1.dept == e2.dept && e1.salary == e2.salary && e1.name <> e2.name)
  [(e1.name, e2.name)]

# QF4 : [String]
(for (t <-- tasks) where (t.task == "abstract")[t.employee])
  ++
(for (e <-- employees) where (e.salary > 50000) [e.name])
```

Figure C.38: Nolineage queries, part 2

*Appendix C.1. Perm comparison*

Table declarations and where-provenance queries in $\mathsf{Links}^{\mathsf{W}}$.

```
var db = database "links";
var i_s_c_o_n_1 =
  table "i_s_c_o_n_1"
  with (oid: Int, i: Int, s: String, cardinal: String, ordinal: String)
  where cardinal prov default tablekeys [["oid"], ["i"]] from db;
...
query {
  for (t_1 <-- i_s_c_o_n_1) ... for (t_m <-- i_s_c_o_n_m)
  where (mod(t_1.i, 100) < 5 && t_1.i == t_2.i && ... && t_1.i = t_m.i)
    [(c1 = t_1.cardinal, c2 = t_2.cardinal, ..., cm = t_m.cardinal)]
}
```

The $\mathsf{Links}^{\mathsf{W}}$ results with where-provenance enabled look something like this with pretty printing of provenance-annotated values disabled (we can see the type Prov(a) really desugars to the tuple type (!data:a, !prov:(String, String, Int))):

```
[(c1=(!data="one",!prov=("i_s_c_o_10000_1", "cardinal", 715924950)),
  c2=(!data="one",!prov=("i_s_c_o_10000_2", "cardinal", 715925958)), ...), ...]
```

*Perm* uses arrays to collect annotations of equal rows. In our query, all rows are different, so these are all singleton arrays.

| c1 | annot_c1 | ... |
|---|---|---|
| two hundred sixty-seven | {public.i_s_c_o_10000_1#cardinal#114040340} | ... |
| three hundred seventeen | {public.i_s_c_o_10000_1#cardinal#114040390} | ... |
| ... | | |

$\mathsf{Links}^{\mathsf{L}}$ lineage queries and part of an example result.

```
lineage {
  query {
    for (t_1 <-- i_s_c_o_n_1) ... for (t_m <-- i_s_c_o_n_m)
    where (mod(t_1.i, 100) < 5 && t_1.i == t_2.i && ... && t_(m − 1).i == t_m.i)
      [(i=t_1.i, c = t_1.cardinal)]
}}

[(data=(c="one", i=1),
  prov=[(row=715924950, table="i_s_c_o_1000_1"),
        (row=715925958, table="i_s_c_o_1000_2"),
        ...]),
 (data=(c="two", i=2),
  prov=[(row=715924951, table="i_s_c_o_1000_1"), ...]),
 ...]
```

The template for "equivalent" *Perm* queries is shown below. We use the **PROVENANCE** keyword which enables Perm influence contribution semantics.

```
SELECT PROVENANCE t_1.i, t_1.cardinal
FROM i_s_c_o_n_1 AS t_1, ..., i_s_c_o_n_m AS t_m
WHERE t_1.i % 100 < 5 AND t_1.i = t_2.i AND ... AND t_(m − 1).i = t_m.i
```