Language-integrated Provenance

Stefan Fehrenbach and James Cheney
University of Edinburgh
stefan.fehrenbach@ed.ac.uk, jcheney@inf.ed.ac.uk

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 heavy-weight 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 paper 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 (Cooper et al. 2007), Hop (Serrano 2009) and Ur/Web (Chlipala 2015) 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 (Meijer et al. 2006), a technique for safely embedding database queries into programming languages.

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, 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 (Cui et al. 2000; Buneman et al. 2001; Green et al. 2007; Glavic and Alonso 2009b). 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, 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 paper, 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 underlying 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# (Syme 2006), SML# (Ohori and Ueno 2011), or Ur/Web (Chlipala 2015).

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 clients.

Our approach builds on Links’s support for queries that construct nested collections (Cheney et al. 2014c). 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. Although we have not yet compared our approach directly to other systems, these results appear to be in a reasonable range: for example, the Perm system (Glavic and Alonso 2009b) reports slowdowns of 3–30 for a comparable form of lineage.

This paper significantly extends an earlier workshop paper (Fehrenbach and Cheney 2015). The workshop version only outlined our initial design for where-provenance in Links; this paper presents the fully-implemented system, extends it to support lineage, and gives a detailed experimental evaluation of both extensions.

2. OVERVIEW

In this section we give an overview of our approach, first covering 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.

Base types $O ::= \text{Int} \mid \text{Bool} \mid \text{String}$

Rows $R ::= \cdot \mid \{ R_1 : A \}$

Table types $T ::= \text{table}(R)$

Types $A, B ::= O \mid T \mid A \rightarrow B \mid \{ R \} \mid \{ A \}$

Contexts $\Gamma ::= \cdot \mid \Gamma, x : A$

Expressions $L, M, N ::= c \mid x \mid \{ i_1 = M_1 \} \cdot N.1\mid \text{fun } f(x_1) \cdot N \mid N(M)$\mid $\text{var } x = M; N \mid \text{if } (L) \{ M \} \text{else } \{ N \}$\mid $\text{query } \{ N \} \mid \text{table } n \text{ with } (l_1; l_2; \ldots)$\mid $[] \mid N \mid N \cdot M \mid \text{empty}(M)$\mid $\text{for } (x \leftarrow L) \cdot M \mid \text{where}(M) \cdot N$\mid $\text{for } (x \leftarrow L) \cdot M \mid \text{insert } L \text{ values } M$\mid $\text{update } (x \leftarrow L) \cdot M \mid \text{set } N$\mid $\text{delete } (x \leftarrow L) \cdot M$,

Figure 3: Syntax of a subset of Links.

We will use a running example of a simple tours database, with some example data shown in Figure 5.

2.1 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 paper. We also omit detailed discussion of the operational semantics of Links, which is presented in previous work (Lindley and Cheney 2012).

Figure 3 presents a simplified subset of Links syntax, sufficient for explaining the provenance translations in this paper. Types include base types $O$ (such as integers, booleans and strings), table types $\text{table}(A_1; A_2)$, function types $A \rightarrow B$, record types $(i_1; A_1)$, and collection types $\{ A \}$. In Links, collection types are treated as multisets inside database queries (reflecting SQL’s default multisemantics), but represented as lists during ordinary execution.

Expressions include standard constructs such as constants, variables, record construction and field projection, conditionals, functions and application. We freely use pair types $(M, N)$ and pair syntax $(M, N)$, which is presented in previous work (Lindley and Cheney 2012).

The expression $\text{query } \{ M \}$ introduces a query block, whose content is translated into 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 $\{ A \}$, and concatenation of collections $M \cdot N$. In addition, the comprehension expressions $\text{for } (x \leftarrow M) \cdot N$ and $\text{for } (x \leftarrow M) \cdot L$ allow us to form queries involving iteration over a collection. The difference between the two expressions is that $\text{for } (x \leftarrow M)$ expects $M$ to be a table reference, whereas $\text{for } (x \leftarrow M)$ expects $M$ to be a collection. The expression $\text{where } (M) \cdot N$ is essentially equivalent to $\text{if } (M) \cdot \{ N \} \text{ else } \{ [] \}$.
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.

The type system (again a simplification of the full system) is illustrated in Figure 4. Many rules are standard; we assume a typing signature Σ mapping constants and primitive operations to types. The rule for query {M} refers to an auxiliary judgment A :: 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:

\[ O :: QType \quad (l_i :: A)_{i=1}^n :: QType \quad A :: QType \]

Similarly, the R :: BaseRow judgment ensures that the types used in a row are all base types:

\[ R :: BaseRow \quad R, l, o :: BaseRow \]

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 (2012) presents 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 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.

2.2 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 (Shar and Tan 2013).

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. Microsoft’s LINQ library, which provides language-integrated query for .NET languages, is a popular feature of C# and F#. 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
Agencies

<table>
<thead>
<tr>
<th>(oid)</th>
<th>name</th>
<th>based_in</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EdinTours</td>
<td>Edinburgh</td>
<td>412 1200</td>
</tr>
<tr>
<td>2</td>
<td>Burns’s</td>
<td>Glasgow</td>
<td>607 3000</td>
</tr>
</tbody>
</table>

ExternalTours

<table>
<thead>
<tr>
<th>(oid)</th>
<th>name</th>
<th>destination</th>
<th>type</th>
<th>price in £</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>EdinTours</td>
<td>Edinburgh</td>
<td>bus</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>EdinTours</td>
<td>Loch Ness</td>
<td>bus</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>EdinTours</td>
<td>Loch Ness</td>
<td>boat</td>
<td>200</td>
</tr>
<tr>
<td>6</td>
<td>EdinTours</td>
<td>Firth of Forth</td>
<td>boat</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>Burns’s</td>
<td>Islay</td>
<td>boat</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>Burns’s</td>
<td>Mallaig</td>
<td>train</td>
<td>40</td>
</tr>
</tbody>
</table>

Figure 5: Example input data

the approach taken in C# (Meijer et al. 2006), SML# (Ohori and Ueno 2011), and Ur/Web (Chlipala 2015). 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. (1995), and has been adopted in languages such as F# (Syme 2006) and Links (Cooper et al. 2007) as well as libraries such as Database-Supported Haskell (Giorgidze et al. 2011).

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 under certain reasonable assumptions, it is possible to normalize these comprehension expressions to a form that is easily translatable to SQL. For example, the following query

```plaintext
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:

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

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 (2009) and Lindley and Cheney (2012)) 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.

### 2.3 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 (1996)) 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 (Cooper 2009; Grust and Ulrich 2013) and queries that construct nested collections (Cheney et al. 2014c). 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.

```plaintext
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:

```plaintext
var q2 = query {
  for (a <--- agencies)
    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])] | [dest = e.destination, type = e.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 (Cheney et al. 2014c) 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 paper 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 (Cooper 2009; Lindley and Cheney 2012; Cheney et al. 2014c), but are not needed to understand our approach.

2.4 Where-provenance and lineage

As explained in the introduction, provenance tracking has been explored extensively for queries 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 paper 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. (2001) introduced this idea; our approach is based on a later presentation for the nested relational calculus by Buneman et al. (2008). 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:

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

The annotation 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 object in the input. The value of the phone field 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:

```scala
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)
}
```

The result of this query is as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>lineage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdinTours 412 1200</td>
<td>(Agencies,1),(ExternalTours,5)</td>
<td></td>
</tr>
<tr>
<td>EdinTours 412 1200</td>
<td>(Agencies,1),(ExternalTours,6)</td>
<td></td>
</tr>
<tr>
<td>Burns’s 607 3000</td>
<td>(Agencies,2),(ExternalTours,7)</td>
<td></td>
</tr>
</tbody>
</table>

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

```scala
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 paper, we give the details of these translations and present an experimental evaluation showing that its performance is reasonable.

3. PROVENANCE TRANSLATIONS

In this section we present the key technical contributions of this paper. We present two extensions of Links: Links\textsuperscript{\textregistered}, which supports where-provenance in queries, and Links\textsuperscript{\textregistered}, which supports lineage in queries. We show that both extensions can be implemented by a type-preserving source-to-source translation to plain Links.
Values of type `Links` without a visible constructor, so only the
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, of type `Prov(Int)` which was copied from row 23,
column a, of table QA. We print the provenance of a value as
a comment (following #) to indicate that it can not be
directly entered into `Links_W`. The type `Prov`, without a
visible constructor, so only the `Links_W` runtime can
construct values of provenance type.

There are two operations on values with provenance type:
`data N` extracts the data value of some expression `N`; similarly,
`prov N` extracts its argument's where-provenance triple.

In addition, 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 `(l : O) ->
(String, String, Int)`. We have three kinds of columns: (1) regular
columns with labels `l_r` where `r` is in some set of inde-
ces `R`. For these columns we do not compute provenance.

(2) Columns with `default` where-provenance have labels `l_d`
where `d` in `D`. For these columns we compute provenance
derived from their location in the database given by table name,
column name, and the row's id. (3) Columns with `external`
where-provenance have labels `l_e` where `e` in `E`.
For these columns we obtain provenance by calling a user-provided
function with the row as input. Such user-defined prove-
nance calculation functions have to be pure and database-
executable, but they are otherwise free to do whatever they
want. The envisioned use is fetching existing provenance
metadata that is stored separately from the actual data.

The typing rules for the new constructs of `Links_W` are shown
in Figure 6. These rules employ an auxiliary judgment `Γ ⊢ S`:
meaning that in context `Γ`, 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 `Prov(A)`, and `data` extracts
its data, the `A`-value. The most complex rule is that for the
table construct. The rule for typing table references also
uses an auxiliary operation `R ⊿ 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.

We give the semantics of `Links_W` by a translation to `Links`.
The syntactic translation of types `Prov[-]` is shown in Fig-
ure 7. We write `Prov[-]` for the obvious extension of the type
translation to contexts. The implementation extends the
Links parser and type checker, and desugars the `Links_W` AST
to a Links AST after type checking, reusing the backend
mostly unchanged. The expression translation function is
also written `Prov[-]` and is shown in Figure 8.

Values of type `Prov(O)` are represented at runtime in `Links`
as ordinary 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 com-
ponent 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 table 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 ⊿ 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: oth-
erwise, 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
\[ \text{if } (L \{M\} \text{ else } \{N\}) = \text{if } (\text{if } (L) \{M\} \text{ else } \{N\}) \]
\[ \text{empty } (\{M\}) = \text{empty } (\{M\}) \]
\[ \text{for } (x \leftarrow L) \{M\} = \text{for } (x \leftarrow \text{if } (L) \{M\} \text{ else } \{N\}) \]
\[ \text{where } (M) = \text{where } (\{M\}) \text{ in } \{N\} \]
\[ \text{for } (x \leftarrow L) \{M\} = \text{for } (x \leftarrow \text{if } (L) \{M\} \text{ else } \{N\}) \]
\[ \text{data } = \{M\}.\text{data} \]
\[ \text{prov } = \{M\}.\text{prov} \]
\[ \text{insert } L \text{ values } \{M\} = \text{insert } \{L\}.\text{values} \{M\} \]
\[ \text{update } (x \leftarrow L) \text{ where } M \text{ set } N = \text{update } (x \leftarrow \text{if } (L) \{M\} \text{ else } \{N\}) \text{ where } \{M\} \text{ set } \{N\} \]
\[ \text{delete } (x \leftarrow L) \text{ where } M = \text{delete } (x \leftarrow \text{if } (L) \{M\} \text{ else } \{N\}) \text{ where } \{M\} \]

Figure 8: Translation of Links\(^W\) to Links, and auxiliary operation \(R \triangleright \Delta S\)

\[ (R, l : \text{Prov}(O)) \triangleright (S, l : \text{prov } \text{default}) = (R \triangleright \Delta S, l = (\text{data } = x.l, \text{prov } = (n, l_j, x.\text{oid})) \]
\[ (R, l : \text{Prov}(O)) \triangleright (S, l : \text{prov} \{M\}) = (R \triangleright \Delta S, l = (\text{data } = x.l, \text{prov } = \{M\}(x)) \]

Figure 9: Doubling and lineage translations

\[ \Gamma \vdash \text{lineage } \{M\} : \text{[}A\text{]} \]

Figure 10: Additional typing rule for Links\(^4\)

### 3.2 Lineage

Links\(^4\) adds the \text{lineage} keyword to Links. The syntax is extended as follows:

\[ L, M, N ::= \cdots \text{ | lineage } \{M\} \]

The expression \text{lineage } \{M\} is similar to \text{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 \text{query } \{M\}). However, instead of executing the query normally, \text{lineage } \{M\} also computes lineage for each record in the result. If \(M\) has type \(\{A\}\) (which must be an appropriate query result type) then the type of the result of \text{lineage } \{M\} will be \(\text{[}A\text{]}\), where \(\text{[}\cdot\text{]}\) is a type translation that adjusts the types of collections \(\{A\}\) to allow for lineage, as shown in Figure 10.

The syntactic translation of Links\(^4\) types is shown in Figure 9. We write \(\text{D}[\Gamma]\) and \(\text{L}[\Gamma]\) for the obvious extensions of these translations to contexts. The translation of Links\(^4\)
expressions to Links is shown in Figure 11. It operates in two modes: \( \mathcal{D} \) and \( \mathcal{L} \). We translate ordinary Links programs using the translation \( \mathcal{D}[-] \). When we reach a \textit{lineage} block, we switch to using the \( \mathcal{L}[-] \) translation. \( \mathcal{L}[[M]] \) provides the lineage for list literals. Their lineage is simply empty. Table comprehension is the most interesting case. We translate a table iteration \( \{ x \leftarrow L \} M \) 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 \( \mathcal{L}[M] \)—the original comprehension body \( M \) transformed using \( \mathcal{L} \). 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 \( \mathcal{L}[M] \) by \( y \). 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 \textit{where}-clauses is straightforward, as shown in Figure 11.)

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 run time) 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 records such as \( \{ \text{data} = 1, \text{prov} = y, \text{prov} \} \) 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 \( \text{d2l}[A] \) accomplishes this by mapping a value of type \( \mathcal{L}[A] \) to one of type \( \mathcal{L}[[A]] \). We define \( \mathcal{L}[-] \) as a function that applies \( \mathcal{L}[-] \) to its argument and substitutes all free variables \( x : A \) with \( \text{d2l}[A](x) \).

The \( \mathcal{D}[-] \) 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 $\mathcal{D}[\cdot]$ translation creates a pair, whose first component is the recursive $\mathcal{D}[\cdot]$ translation of the function, and whose second component uses the $\mathcal{L}'[\cdot]$ translation to create a version of the function callable from within a lineage block. (We use $\mathcal{L}'[\cdot]$ 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 $A \rightarrow B$ case of $\text{dil}$.)

Finally, notice that the $\mathcal{D}[\cdot]$ translation maps table types and table references to pairs. This is similar to the $\mathcal{D}[\cdot]$ 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:

```plaintext
for (y_a <- agencies.2())
  for (z_a <- for (y_e <- externalTours.2())
    (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, Link’s built-in normalization algorithm simplifies this query to:

```plaintext
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),
     ("Agencies","y_a.oid"), ("ExternalTours","y_e.oid"))]
```

The (again, intended) correctness property for the translation from Link$^w$ to Link$^s$ is stated as follows:

**THEOREM 2.** Let $M$ be given such that $\Gamma \vdash_{\text{Link}} M : A$. Then:

1. $\mathcal{L}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
2. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
3. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{D}'[M] : \mathcal{D}[A]

The proof of each part is straightforward by induction (notice that $\mathcal{D}[\cdot]$ depends on $\mathcal{L}[\Gamma]$ but not vice versa). The main complication is the use of $\mathcal{L}'[\cdot]$ inside such blocks are translated to project out the second component (this is actually accomplished via the $A \rightarrow B$ case of $\text{dil}$.)

The (again, intended) correctness property for the translation from Link$^w$ to Link$^s$ is stated as follows:

**THEOREM 2.** Let $M$ be given such that $\Gamma \vdash_{\text{Link}} M : A$. Then:

1. $\mathcal{L}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
2. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
3. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{D}'[M] : \mathcal{D}[A]

The proof of each part is straightforward by induction (notice that $\mathcal{D}[\cdot]$ depends on $\mathcal{L}[\Gamma]$ but not vice versa). The main complication is the use of $\mathcal{L}'[\cdot]$ inside such blocks are translated to project out the second component (this is actually accomplished via the $A \rightarrow B$ case of $\text{dil}$.)

The (again, intended) correctness property for the translation from Link$^w$ to Link$^s$ is stated as follows:

**THEOREM 2.** Let $M$ be given such that $\Gamma \vdash_{\text{Link}} M : A$. Then:

1. $\mathcal{L}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
2. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
3. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{D}'[M] : \mathcal{D}[A]

The proof of each part is straightforward by induction (notice that $\mathcal{D}[\cdot]$ depends on $\mathcal{L}[\Gamma]$ but not vice versa). The main complication is the use of $\mathcal{L}'[\cdot]$ inside such blocks are translated to project out the second component (this is actually accomplished via the $A \rightarrow B$ case of $\text{dil}$.)

The (again, intended) correctness property for the translation from Link$^w$ to Link$^s$ is stated as follows:

**THEOREM 2.** Let $M$ be given such that $\Gamma \vdash_{\text{Link}} M : A$. Then:

1. $\mathcal{L}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
2. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{L}'[M] : \mathcal{L}[A]
3. $\mathcal{D}[\Gamma] \vdash_{\text{Link}} \mathcal{D}'[M] : \mathcal{D}[A]

The proof of each part is straightforward by induction (notice that $\mathcal{D}[\cdot]$ depends on $\mathcal{L}[\Gamma]$ but not vice versa). The main complication is the use of $\mathcal{L}'[\cdot]$ inside such blocks are translated to project out the second component (this is actually accomplished via the $A \rightarrow B$ case of $\text{dil}$.)
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, client: String)

Figure 12: Benchmark database schema, c.f. Cheney et al. (2014c).

# Q1: [(contacts*: Prov(Boolean), 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 e.dept) == (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 = getOutliers(x.employees),
     fun (y) { map(fun (z) { data z }, y.tasks) } ++
     getContacts(x.contacts),
     fun (y) { [ "buy" ] } )]

Figure 13: “allprov” benchmark queries used in experiments

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,
but just return less type-safe. Q3 checks provenance from
the employee. Q4 returns the employees’ provenance only,
and 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 Links\(^W\) 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 Links\(^W\) 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 Links\(^W\) built-in function serverTimeMillis which
in turn uses OCaml’s Unix gettimeofday.

Data. Figure 14 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”.

Figure 14: Where-provenance query runtimes.

<table>
<thead>
<tr>
<th>Query</th>
<th>median runtime(^{*}) in ms</th>
<th>overall slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>allprov</td>
<td>someprov</td>
</tr>
<tr>
<td>Q1</td>
<td>6068</td>
<td>3653</td>
</tr>
<tr>
<td>Q2</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Q3</td>
<td>8100</td>
<td>8064</td>
</tr>
<tr>
<td>Q4</td>
<td>1502</td>
<td>1214</td>
</tr>
<tr>
<td>Q5</td>
<td>6778</td>
<td>3457</td>
</tr>
<tr>
<td>Q6</td>
<td>17874</td>
<td>18092</td>
</tr>
</tbody>
</table>

Figure 15: Median runtimes for largest dataset (Q1
at 512 departments, Q5 at 1024 departments, Q6
at 2048 departments, others 4096 departments) and
global means of overall slowdowns
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 size 4 for the mean slowdown in QF4 which reported taking geometric mean over all database sizes. (We exclude database also reports the slowdown of lineage versus no lineage as the geometric mean for the largest database instance that both queries are in black circles, no lineage is shown as yellow circles.) Measurements with data.

Figure 17 shows our lineage experiment results. Query Q7 ran out of memory for larger databases. We also report the slowdown of lineage versus no lineage as the geometric mean, for the largest database instance that both queries are in black circles, no lineage is shown as yellow circles.) Measurements with data.

4.2 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 Lineage we only use databases up to 1024 departments, because most of the queries are a lot more expensive. Query QCs are in black circles, no lineage is shown as yellow circles.) Measurements with data.

Figure 16 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. 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. (2014c). 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 17 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 18 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

type name Lin(a) = (data: a. prov: [(row: Int, "table": String)]);

# AQ6: [Lin((department: String, outliers: [Lin(name: String, ... for (d <= dept) => for (e <= employees) where (department == e.dept) + [name = e.name, salary == e.salary], name == d.name)])

# 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 (department == 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) [t.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 < 10000)]

[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 == "o", 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 16: Lineage queries used in experiments
The experiments confirm this. Lineage is still somewhat in memory. This should reduce the overhead (in terms of particular has a large memory overhead. In practice, for representation of values in general and database results in the whole database in a different shape.

4.3 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.

5. RELATED WORK

Buneman et al. (2001) gave the first definition of where-provenance in the context of a semistructured data model. The DBNotes system of Bhagwat et al. (2005) 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. (2008) 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 LinksW 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. (2008) 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.

Links' is inspired by prior work on lineage (Cui et al. 2000) and why-provenance (Buneman et al. 2001). 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. (2008) 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 (2009b) introduced the Perm system, which translated ordinary queries to queries that compute their own lineage; they handled a larger sub-
language of SQL than previous systems such as Trio, and
subsequently Glavic and Alonso (2009a) extended this ap-
proach 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.
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 LinksW and Links² rely on the conservativity and query
normalization results that underly Links’s implementation of
language-integrated query, particularly Cooper’s work (2009)
conservativity to queries involving higher-order functions,
and previous work by Cheney et al. (2014c) 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 Grust et al.’s loop-lifting
(2010) and more recent work on query flattening (Urich and
Grust 2015), and it would be interesting to evaluate the
performance of these techniques on provenance queries.

Other authors, starting with Green et al. (2007), 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 (Amsterdamer et al. 2011). Kar-
vounarakis et al. (2010) developed ProQL, an implementa-
tion of the semiring model in a relational database via SQL
query extensions. Glavic et al. (2013) 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. We
believe that semiring polynomial annotations can also be ex-
tacted from lineage in Links, but supporting other instances
of the semiring model via query rewriting in Links appears
to be nontrivial due to the need to perform aggregation. In
future work, we intend to increase the expressiveness of Links
queries to include aggregation and grouping operations and
strengthen the query normalization results accordingly.

LinksW and Links², 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 have not yet investigated this and it is
not clear whether it is straightforward or difficult; one
possible difficulty may be the need to combine multiple type
translations. We intend to explore this (as well as consider
alternative models). Cheney et al. (2014a) presented a gen-
eral 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 repre-
sent the traces in this approach in a relational data model.
(Giorgidze et al. (2013) 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. (2014b) 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.

6. CONCLUSIONS
Our approach shows that it is feasible to implement pro-
venance 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 normali-
azation 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 ap-
proach 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.

Links is a research prototype language, but the underly-
ing ideas of our approach could be applied to other sys-
tems 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. Our results
show that provenance for database queries can be imple-
mented efficiently and safely at the language-level. This
is a promising first step towards systematic programming
language support for provenance.

References
Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for
O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald,
and J. Widom. Databases with uncertainty and lineage.
An annotation management system for relational databases.
P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Princi-
ples of programming with complex objects and collection
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 Berlin /
P. Buneman, J. Cheney, and S. Vansummeren. On the
expressiveness of implicit provenance in query and update
J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in
databases: Why, how, and where. Foundations and Trends
J. Cheney, A. Ahmed, and U. A. Acar. Database queries
that explain their work. In PPDP 2014, pages 271–282.
J. Cheney, S. Lindley, G. Radanne, and P. Wadler. Effective
quotations: Relating approaches to language-integrated