Language-integrated Provenance

Citation for published version:

Digital Object Identifier (DOI):
10.1145/2967973.2968604

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
PPDP '16 Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Language-integrated Provenance

Stefan Fehrenbach  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 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 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) 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...
We first review a subset of the Σ addition, equality tests, etc.; their types are collected in a signature c definable using records. Constants (tions and application. We freely use pair types (such as integers, booleans and strings), table types (such as 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 Σ. In Links we write var x = M; N for binding a variable x to M in a N. The semantics of the Links constructs 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 essentially 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.

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 their 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:

\[ A :: QType \quad \text{where} \quad \frac{[A_i :: QType]_{i=1}^n}{A :: QType} \]

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

\[ R :: \text{BaseRow} \quad \frac{R, i : O :: \text{BaseRow}}{\text{BaseRow}} \]

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 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.

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 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

\[
\text{var q1 = query \{ }
\]
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 (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.

```
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` (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:

```
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 row. The field value will be of type `Prov(String)`; the value data 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)]
}
```

The result of this query is as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>p_phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdinTours</td>
<td>412 1200</td>
<td>(Agencies, phone, 1)</td>
</tr>
<tr>
<td>EdinTours</td>
<td>412 1200</td>
<td>(Agencies, phone, 1)</td>
</tr>
<tr>
<td>Burn’s</td>
<td>607 3000</td>
<td>(Agencies, phone, 2)</td>
</tr>
</tbody>
</table>

`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 (Cui et al. 2000; Buneman et al. 2001; Cheney et al. 2009; Glavic et al. 2013), however, many of these only make sense for set-valued collections, whereas Links currently supports multiset semantics. In this paper, 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 different definitions of why-provenance (for example, to require minimality).

A common approach is to define 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 \((\text{AgencyTours}, 2)\) 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 \( q_1 \) is as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>row</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdinTours</td>
<td>412 1200</td>
<td></td>
</tr>
<tr>
<td>EdinTours</td>
<td>412 1200</td>
<td>23</td>
</tr>
<tr>
<td>Burns’s</td>
<td>607 000</td>
<td>23</td>
</tr>
</tbody>
</table>

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{W}, which supports where-provenance in queries, and Links\textsuperscript{L}, which supports lineage in queries. We show that both extensions can be implemented by a type-preserving source-to-source translation to plain Links.

3.1 Where-Provenance

Links\textsuperscript{W} extends Links with support for where-provenance. The syntax shown in Figure 3 is extended as follows:

\[
\begin{align*}
O &::= \cdots | Prov(O) \\
L, M, N &::= \cdots | data M | prov M | table(R) where S \\
S &::= \cdot | S, l prov s \\
s &::= default | M
\end{align*}
\]

We introduce the type constructor \( Prov(O) \), where \( O \) is a type argument of base type. We treat \( Prov(O) \) 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 \( Prov(O) \) are annotated with where-provenance, 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, \( 42 \neq \#(“QA”, “a”, 23) \) represents the row 42, of type \( Prov(\text{Int}) \) 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 cannot be directly entered into Links\textsuperscript{W}. The type \( Prov(O) \) is abstract, without a visible constructor, so only the Links\textsuperscript{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 \((l, O) \rightarrow (\text{String, String, Int})\). We have three kinds of columns: (1) regular columns with labels \( l_i \) where \( r \) is in some set of indices \( R \). 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 oid. (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 provenance 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\textsuperscript{W} are shown in Figure 6. These rules employ an auxiliary judgment \( \Gamma \vdash S : ProvSpec(R) \), meaning that in context \( \Gamma \), the provenance specification \( S \) is valid with respect to record type \( R \). As suggested by the typing rule, the \text{prov} keyword extracts the provenance from a value of type \( Prov(A) \), and \text{data} extracts its data, the \( A \)-value. The most complex rule is that for the \text{table} construct. The rule for typing table references also uses an auxiliary operation \( R \bowtie S \) that defines the
We construct a new record with the same fields as the table. For simplicity, the keywords are represented at runtime in the above types is shown in Figure 9. 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_prov = 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 a single SQL query.

The (intended) correctness property of the where-provenance translation is that it preserves well-formedness, as follows:

**Theorem 1.** For every \(\text{Links}^W\) term \(M\):

\[
\Gamma \vdash \text{Links}^W M : A \Rightarrow \text{Links}^W [M] : \text{Links}^W [A]
\]

The proof is straightforward by induction on the structure of derivations; the only interesting cases are those for comprehensions and updates, since they illustrate the need for both the plain table reference and its provenance view.

### 3.2 Lineage

Links\(^L\) adds the **lineage** keyword to Links. The syntax is extended as follows:

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

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** \(\{ M \}\)). However, instead of executing the query normally, **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 **lineage** \(\{ M \}\) will be \(\mathcal{L}[A]\), where \(\mathcal{L}[-]\) 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\(^L\) types is shown in Figure 9. We write \(\mathcal{D}[\Gamma]\) and \(\mathcal{L}[\Gamma]\) for the obvious extensions of these translations to contexts. The translation of Links\(^L\) expressions to

\[
\text{Table } n \text{ with } (R) \text{ where } S = (\text{table } n \text{ with } (R), \text{fun}() \{ \text{for } x <\in \text{ table } n \text{ with } (R) \{ (R \, x \, n) \} \})
\]
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 lineage block, we switch to using the $\mathcal{L}[-]$ translation. $\mathcal{L}[M]$ provides initial lineage for list literals. Their lineage is simply empty. Table comprehension is the most interesting case. We translate a table iteration for $(x <\cdots L) M$ 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 $\mathcal{L}[e]$ 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 where-clauses is straightforward, as shown in Figure 11.)

One subtle 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 $(data = 1, prov = L)$ 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 $d2[[A]]$ accomplishes this by mapping a value of type $\mathcal{D}[A]$ 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 $d2[[A]](x)$.

The $\mathcal{L}[-]$ 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 $\mathcal{D}[-]$ translation creates a pair, whose first component is the recursive $\mathcal{D}[-]$ translation of the function, and whose second component uses the $\mathcal{L}[-]$ translation to create a version of the function callable from within a lineage block. (We use $\mathcal{L}[-]$ 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 $A \rightarrow B$ case of $d2l$.)

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

The (again, intended) correctness property for the translation from Links$^2$ to Links is stated as follows:

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

1. $\mathcal{L}[\Gamma] = \text{Links} \mathcal{L}[M] : \mathcal{L}[A]$
2. $\mathcal{D}[\Gamma] = \text{Links} \mathcal{D}[M] : \mathcal{D}[A]$
3. $\mathcal{D}[\Gamma] = \text{Links} \mathcal{D}[M] : \mathcal{D}[A]$

The proof of each part is straightforward by induction (notice that $\mathcal{D}[-]$ depends on $\mathcal{L}[-]$ but not vice versa). The main complication is the use of $l2s$ in $\mathcal{L}[-]$, and the cases for functions and lineage which need to use the second induction hypothesis. In the case of lineage, we use the fact that $\mathcal{D}[A] = \mathcal{D}[\mathcal{L}[A]]$, which follows because $A :: QType$ so cannot involve table or function types.

### 4. Experimental Evaluation

We implemented two variants of Links with language-integrated provenance, Links$^2$ and Links$^3$, featuring our extensions for where-provenance and lineage, respectively. Both variants build on Links with query shredding as described by Cheney et al. (2014c); they used queries against a simple test database schema (see Figure 12) that models an organization with departments, employees and external contacts. We change some of their benchmarks to return where-provenance and provenance and compare against the same queries without provenance.

Unlike Cheney et al. (2014c) 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. (2014c) describe it: “We vary the number of departments in the organisation 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 quad-core CPU with 3.2 GHz, 8 GB RAM, and a 500 GB
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 whenever 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 complexity 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. (2014c) 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. This changes the types, so we have to adapt the queries and some of the helper functions. Figure 13 shows the benchmark queries with full provenance. 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 with a fake task "buy". Since the fake task is not a database value it cannot have where-provenance. Links9 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
allprov


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

# Q2 : [(d: Prov(String))]
for (d <= q1(1))
where (all(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 : [(dept: Prov(String), emps: [Prov(String)])]
for (d <= departments)
[(dept = 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(1))
[(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 13. “allprov” benchmark queries used in experiments

Figure 14. Where-provenance query runtimes.

Figure 15. Median runtimes for largest dataset (Q1 at 512 departments, Q5 at 1024 departments, Q6 at 2048 departments, others 4096 departments) and geometric means of overall slowdowns

The table in Figure 15 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 15 also reports the slowdown of full where-provenance versus no provenance as the geometric mean over all database sizes, for each query. The slowdown ranges from 1.22 for query Q6 up to 2.8 for query Q4.

Interpretation. The graphs suggest that the asymptotic complexity of all three variants is the same, confirming our hypothesis. This was unexpected, 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 for some database sizes, because the queries returned results that were too large for Links to construct as in-memory values.

The table...
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 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 complexity 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 16 lists the queries used in the lineage experiments. For lineage queries, they 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 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.

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 Links\(^\d\) 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\(^5\) 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 sublanguage of SQL than previous systems such as Trio, and subsequently Glavic and Alonso (2009a) 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. 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 Links\(^5\) and Links\(^6\) rely on the conservativity and query normalization results that underly Link’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. (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 (Ulrich 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 (Amsterdem et al. 2011). Karvounarakis et al. (2010) developed ProQL, an implementation 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 extracted 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. Links\textsuperscript{6} and Links\textsuperscript{5} 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 intend to explore this (as well as consider alternative models). Cheney et al. (2014a) 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. (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 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.

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. 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.

References


A. Benchmark code

This appendix contains the full listings for the where-provenance and lineage benchmarks. Figures 19 and 20 show the plain table declarations and declarations with where-provenance, respectively. These tables also include readonly and tablekeys annotations which were suppressed in the paper; the former indicates that a field is read-only and the latter lists the subsets of the fields that uniquely determine the others.

Figure 21 shows the helper functions used by the plain versions of the queries, and Figure 22 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 23 shows the versions of the queries with some provenance (the someprove benchmarks).

Figure 24 shows the plain queries without lineage annotations; these also employ abbreviations from Figure 21.

```plaintext
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 19. 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 20. Table declarations for where-provenance queries (except noprov).
```
Figure 21. Helper functions noprov.

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

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

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

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

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

sig clients: [[["client": Bool|a]] -> [["client": Bool|a]]
  fun clients(xs) {
    filterfun (x) (data."client"). Is Poor (put data in a bunch of 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),
names = 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 : () -> [(dept: Prov(String), emps: [String, String, Int])] run q4() {
for (d <- departments)
[(dept = d.name),
emps = for (e <- employees)
where ([(data d.name) = (data e.dept))
[prov e.name()]]
}

# Q5: Tasks with employees and departments

fun dropProv() {
map(fun (x) { data x }, [1])
}

# Only get provenance of tasks, drop other provenance. Reuses Q3: employeesByTask, which still has provenance types, but does not actually compute all provenance.

sig q5 : () -> [(a: Prov(String),
b: [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 Deproven on department.

sig q6 : () -> [(department: String, people: [name: Prov(String), tasks: [String]])]

fun q6() {
for (x <- q_org())
[(department = data x.name, people = getOutliers(x.employees),
fun (y) { map(fun (z) { data z }, y.tasks)) ++
getLambda(x.contacts),
fun (y) { ["buy"] }])
}

---

Figure 23. Queries someprov.

---

# AQ5 : [(department: String, outliers: [name: String, salary: Int, tasks: [String]]]

for (d <- departments)
[(employees = for (e <- employees)
where (d.name == e.dept)
[(name = e.name, salary = e.salary),
names = d.name)])
outliers = for (o <- d.employees)
where (o.salary > 1000000 || o.salary < 1000)
[oj])

# Q3 : [(b: [String]), e: String]]

for (e <- employees)
[(b = tasksOfEmp(e), e = (e.name))]

# Q4 : [(dept: String, emps: [String]])

for (d <- departments)
[(dept = d.name),
emps = for (e <- employees)
where (d.name == e.dept)
[(e.name)]]

# Q5 : [(a: String, b: [name: String, sal: 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 > 100000)])
[(name = y.name, tasks = for (z <- tasks)
where (z.employee = y.name)
[([z.task])]++)
for (t <- contacts)
where (x.name = y.dept && x."client")
[(name = y.dept, tasks = ["buy"]))]])

# 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 && y.name == x.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 (el <- employees)
for (e2 <- employees)
where (el.dept == e2.dept && el.salary == e2.salary && el.name <> e2.name)
[(el1.name, e2.name)]

# QF4 : [String]

for (x <- tasks)
where (t.task == "abstract")
[t.employee]++
for (e <- employees)
where (e.salary > 50000)
[e.name]

---

Figure 24. Nolineage queries