Language-integrated Provenance in Links

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

Abstract

Today's programming languages provide no support for data provenance. In a world that increasingly relies on data, we need provenance to judge the reliability of data and therefore should aim for making it easily accessible to programmers. We report our work in progress on an extension to the Links programming language that builds on its support for language-integrated query to support *where*-provenance queries through query rewriting and a type system extension that distinguishes provenance metadata from other data. Our approach aims to work solely within the language implementation and thus require no changes to the database system. The type system together with automatic propagation of provenance metadata will prevent programmers from accidentally changing provenance, losing it, or misattributing it to other data.

1. Introduction

Provenance information, knowing where data came from, is crucial in a variety of situations. Over the last few years, researchers extended databases to store, query, and propagate provenance information. Such techniques are strongly motivated by applications to data integrity and authenticity in open, collaborative settings, such as Web databases or data integration systems where data are combined from numerous sources of variable quality.

At the same time, advances in language-integrated query bridge the gap between relational databases and programming languages [9, 16]. However, so far these avenues of research seem to take little notice of each other. In particular, there has been little investigation of how to write programs that interact with a database that provides support for provenance. Most applications where provenance is needed do not just involve a database in isolation; for example, a typical web application usually involves at least three separate layers, each controlled by a different language such as server-side middleware (e.g. Java, Python, PHP), database (SQL), and Web client (JavaScript). We think the interaction of programming languages with provenance can be greatly improved by leveraging ideas from language-integrated query and Web programming.

We extend the research programming language Links [9] with explicit support for provenance. Links was originally designed to support writing three tier web applications in a single language. Links programs can run on the client, the server, and query the database using list comprehensions that translate to SQL; furthermore, its type system statically checks that the communication between these layers is well-behaved. Links provides type-safe language-integrated querying, as described elsewhere [8, 15]. In particular, the type system of Links checks statically that embedded query expressions will generate a single SQL query.

To illustrate, imagine the following scenario: Consider Pear Computers, a major international company specializing in high-end music players, smartphones, and smartwatches. We are developing a website on behalf of Pear Computers that features customers' comments to promote products. Figure 1 illustrates this scenario using Links. We have a table `top_comments` that aggregates comments on all Pear Computers products from a variety of sources and somehow keeps track of where every comment came from. The **table** declaration tells Links about the existence of the table, and the names and types of its columns. On the product page for Pear's new and highly anticipated smartwatch, the *pWatch*, we only want to show comments that relate to watches, not phones or other products. The `watch_comment` function identifies comments as relating to watches by a crude heuristic: they originate from a table called *watch* or contain the word *pWatch*. We render a single quote with the `render_quote` function which makes use of Links' literal XML support. The `quotes_list` function ties it all together. It queries the database with a **query** block that selects the text of every comment in `top_comments` that is a watch comment. The **query** is guaranteed to generate a single SQL query at run time: it calls a function `watch_comment` that only performs operations that are allowed on the database[1]. Finally, it returns the XML representation of a list of all the watch comments, which can be embedded in an HTML page by other parts of the program.

Now suppose that we would like to be able to remove comments that do not comply with the Pear Computers company policy of only ever saying how innovative and awesome their products are. In the admin panel, we would like to render comments with a *delete* button next to them. Obviously, we have to track where the comments came from to be able to delete them. Also, keep in mind that we are dealing with a legacy system in our example: The `top_comments` table is automatically generated, so it would not help us to delete rows from there. Instead, we have to delete the original data. Fortunately, someone thought ahead and included the necessary information in the `origin_*` columns. However, to add the *delete* button in Links we would now need to change several parts of the program to propagate the origin information to the place where it is needed.

This seems like a perfect fit for *where*-provenance [2–4], in which each data value in the result of a query carries an annotation consisting of either a reference to the source of the data in the input (e.g. a triple containing the table name, field name and row id), or a placeholder called "bottom" ($\bot$) indicating that the value was

[1] Links contains two different function types, to distinguish functions that can be run only on the server (`A ~> B`) from functions that can be run on the database or server (`A -> B`).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

TaPP 2015, July 8–9, 2015, Edinburgh, Scotland.
Copyright remains with the owner/author(s).

```
var top_comments = table "top_comments" with
        (id: Int, text: String,
         origin_table: String, origin_column: String, origin_row: Int);

sig watch_comment : ((text:String, origin_table:String|_)) -> Bool
fun watch_comment(c) {
  c.origin_table == "watch" || c.text =~ /.*pWatch.*/
}

sig render_quote : (String) ~> Bool
fun render_quote(c) {
  <li>
    <blockquote>{stringToXml(c)}</blockquote>
  </li> }

sig quotes_list : () ~> Xml
fun quotes_list() {
  var comments = query {
    for (c <-- top_comments)
    where (watch_comment(c.text))
      [(text = c.text)]
  }
  <ul>{for (c <- comments) render_quote(c.text)}</ul>
}
```

Figure 1. Render a list of quotes from the database.

created by the query. We propose to extend Links to explicitly support provenance, and call the resulting language *PLinks*. In *PLinks* we can implement delete buttons using where-provenance as shown in Figure 2. The **prov**-part of *PLinks*' **table** declaration contains information on how to compute provenance for columns. This is what we use as the text column's provenance.

Having declared how to compute provenance, we can change the type of watch_comment to accept a value of type Prov String, which indicates that we expect a string argument that carries provenance. The keyword **prov** gives access to provenance. It returns a record with three fields: relation, column, and row that correspond to the three components of a where-provenance triple. As before, we consider comments to be about a watch, if they originate from a relation called *watch* (now accessed using (**prov** c).relation) or the text itself (accessed via the keyword **data**) contains the word *pWatch*. We add a new function named delete_quote which uses the text's provenance to delete it from the original table.[2] This function is called when admins click the delete button that is emitted in render_quote.

We could write a program to do the same thing in plain Links. In fact, in Section 3.2 we describe how to translate *PLinks* programs into Links programs. However, language-integrated provenance has some benefits over handling provenance manually or in the database: Provenance is not data. Provenance is metadata. A provenance-aware type system like the one we describe in Section 3.1 ensures it is handled accordingly. Programmers who have a value with provenance type can be certain that it carries provenance that ties it back to its origin in a database. Provenance can never be lost, invented, or arbitrarily manipulated on the way. Precise types can restrict the operations on data to those that are meaningful in the presence of where-provenance, thus eliminating the need for placeholders for data of unknown provenance (or so-called "bottom" values). Language-integrated query already gives uniform access to data in the programming language and data from the database. We extend that to natural access to provenance. Glavic et al. showed that implementing provenance through rewriting queries in the database is possible [12, 13]. If we rewrite programs before interaction with

[2] We use table_from_name to get the actual table from its name.

```
var top_comments = table "top_comments" with
        (id: Int, text: String,
         origin_table: String, origin_column: String, origin_row: Int)
        prov (text = fun (c) { (relation = c.origin_table,
                                column = c.origin_column,
                                row = c.origin_row) });

sig watch_comment : (Prov String) -> Bool
fun watch_comment(c) {
  (prov c).relation == "watch" || data c =~ /.*pWatch.*/
}

sig delete_quote : (Prov String) ~> ()
fun delete_quote(c) server {
  delete (r <-- table_from_name((prov c).relation)
  where (r.id == (prov c).row) }

sig render_quote : (Prov String) ~> Bool
fun render_quote(c) {
  <li>
    <blockquote>{stringToXml(data c)}</blockquote>
    <button l:onclick="{delete_quote(c)}">delete</button>
  </li> }

sig quotes_list : () ~> Xml
fun quotes_list() {
  var comments = query {
    for (c <-- top_comments)
    where (watch_comment(c.text))
      [(text = c.text)]
  }
  <ul>{for (c <- comments) render_quote(c.text)}</ul>
}
```

Figure 2. Programming with provenance support.

the database system, we get provenance support for any unmodified relational database management system for free.

This paper describes work in progress. In the next section, we describe our technical strategy and summarize experience with a preliminary implementation. Section 3 outlines our revised design based on this experience. Section 4 describes related work, Section 5 details limitations and our future plans, and Section 6 concludes.

2. Technical approach and preliminary results

Query expressions in Links are based on the nested relational calculus [5], a core query language that provides collection types and comprehensions:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid (e_1, e_2) \mid e.i \mid e_1 + e_2 \mid e_1 = e_2 \mid \cdots \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \mid & \emptyset \mid e_1 \cup e_2 \mid \{e\} \mid \texttt{for}\ (x \leftarrow e)\ \texttt{return}\ e' \\
\tau & ::= & b \in \{\texttt{int}, \texttt{bool}, \ldots\} \mid t_1 \times t_2 \mid \{t\}
\end{array}
$$

Links queries use a similar syntax (extended with records) and uses a normalization algorithm (explained by Cooper [8] and by Lindley and Cheney [15]) to turn such query expressions into SQL, provided the queries return flat records of values of base types. (Recent work on *query shredding* [6] shows how to lift this restriction, but hasn't yet been incorporated into the main version of Links.) Buneman et al. [3] proposed a translation that maps an ordinary nested relational query to one that propagates where-provenance information on all parts of the source data to the output. We present a simplified version of this translation: given a query expression e, we define $P(e)$ to be a query that propagates annotations on data values of base types

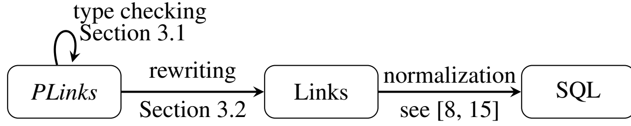

Figure 3. From *PLinks* to SQL.

(integers, strings etc.) only, as follows:

$$
\begin{array}{rcl}
P(x) & = & x \\
P(c) & = & (c, \bot) \\
P(e_1 \; op \; e_2) & = & (P(e_1).1 \; op \; P(e_2).1, \bot) \\
 & & \quad op \in \{+, =, \ldots\} \\
P(e.i) & = & P(e).i \\
P((e_1, e_2)) & = & (P(e_1), P(e_2)) \\
P(\texttt{if}\; e \;\texttt{then}\; e_1 \;\texttt{else}\; e_2) & = & \texttt{if}\; P(e).1 \;\texttt{then}\; P(e_1) \;\texttt{else}\; P(e_2) \\
P(\emptyset) & = & \emptyset \\
P(e_1 \cup e_2) & = & P(e_1) \cup P(e_2) \\
P(\{e\}) & = & \{P(e)\} \\
P(\texttt{for}\; (x \leftarrow e) \;\texttt{return}\; e') & = & \texttt{for}\; (x \leftarrow P(e)) \;\texttt{return}\; P(e')
\end{array}
$$

The translation changes the types of the expressions as follows:

$$T(b) = b \times tag \quad T(\tau_1 \times \tau_2) = T(\tau_1) \times T(\tau_2) \quad T(\{\tau\}) = \{T(\tau)\}$$

where tag stands for the type of the provenance information, e.g. triples of relation name, field name and row id. Given a query expression $e : \tau$ with free variables $x_1 : \tau_1, \ldots, x_n : \tau_n$, its translation $e' = P(e)$ has type $T(\tau)$ assuming the free variables' types are adjusted to $x_1 : T(\tau_1), \ldots, x_n : T(\tau_n)$.

We have used this translation as the basis of an initial attempt to extend Links with provenance support. We modified the Links query normalizer to perform the above translation on queries. However, this approach seems to have some limitations that make it difficult to combine provenance-aware queries with Links programs. For example, since the translation changes the types of both the inputs (e.g. table references) and results, using the translated query requires fixing some convention for naming the additional provenance fields of the tables. These must be added somehow to the database, which imposes a space and maintenance overhead; moreover, the translation requires annotations on all source data, and propagates these annotations through the query eagerly, even if the user never requests them. Moreover, Buneman et al.'s approach relies on a special "bottom" value $\bot$ to annotate parts of the result that were not copied from the input but were instead created by the query. Finally, Buneman et al.'s translation treats the provenance information as pure metadata: it cannot be inspected by the query, nor can it be manipulated or "forged". However, once the provenance leaves the query and is processed by the Links program, it becomes plain data: the programmer can discard it, change it or fabricate it. Both of these choices are questionable: it may be advantageous to be able to inspect the provenance during a query, and it may also be advantageous to be able to certify the integrity of the provenance information (i.e. unforgeability) outside of the query.

These observations motivate a design that takes the eventual use of the provenance information elsewhere in Links programs into account, which will be the focus of the next section.

3. Design of *PLinks*

We describe *PLinks*, an extension of Links, that turns the programming language into a provenance system in its own right. That is, not only will it allow programmers to query provenance from an external database with provenance support, but it will rewrite queries to calculate their own provenance before they even reach the database.

This is work in progress and has not been implemented yet. Figure 3 shows the compilation and execution model. We type check *PLinks* programs using the type system extension described in Section 3.1. We translate type correct programs to standard Links using the source to source translation described in Section 3.2. From there, Links uses its standard query normalization strategy described by Cooper and by Lindley and Cheney [8, 15], to generate SQL queries which can then be executed on any relational database.

The new language features of *PLinks* compared to Links are as follows. *PLinks* adds a new type constructor Prov o, where o is a base type, to represent values that carry where-provenance. To access provenance, we add the keyword **prov**. Given a value of type Prov o, it returns a record representing the value's provenance. Such a provenance record has three fields: relation, column, and row that reflect the first, second, and third component of a where-provenance triple, respectively. The related keyword **data** drops provenance, returning just the data. We used most of these in the example in Figure 2. Values of type Prov o can not be constructed directly by the programmer. Instead, they are automatically generated by *PLinks* (based on the **prov** clauses associated with table definitions). The programmer only specifies how the provenance for a column is derived. This is more flexible than in our initial attempt but possibly not quite enough yet. See Section 5 for a discussion of limitations.

3.1 Type system

Links is a statically typed language. It has *base types* Int, Bool, and String that correspond to their equivalents in databases. Records have row types that describe the labels and types of the record's fields. Row polymorphism allows the same function to operate on differently shaped records, as long as the necessary labels are present. Effect types specify where function can be executed: on the client, the server, or in database queries. The most important aspect of Links' type system, for our purposes, is the guarantee that **query** blocks get translated to a single SQL query if they use only database-executable functions and return a flat list of records with fields of base type. Lindley and Cheney [15] describe Links' type system in detail.

PLinks extends Links' type system to account for where-provenance. Where-provenance is attached to a "cell" in a relation and describes where the data was copied from. Provenance information has meaning only in the context of the data it belongs to. The type system should capture the special nature of provenance metadata. This allows programmers to rely on the accuracy of provenance and prevents mistakes. There are four things in particular, that we expect from a type system that properly deals with provenance: First, provenance is attached to data and automatically propagated with the data it belongs to. Second, provenance is immutable, thus the type system should prevent accidental modification. Third, changes to the data would invalidate where-provenance. Data in Links is immutable, so this problem does not directly manifest, but it may require caution when data in the database is updated. Fourth, it is not possible for a programmer to forge provenance. Provenance is always and only automatically derived from the database by *PLinks*.

Our type system design realizes these goals as follows. The type constructor Prov indicates data paired up with its provenance. Values of type Prov o, where o is a *base type*, carry provenance metadata. The restriction of the type argument o to base types reflects that where-provenance is attached to "cells", not whole database rows. This does not apply to some other forms of provenance (e.g. *why*-provenance [4] based on row annotations), so we will need to revisit this in future work.

Figure 4 shows simplified versions of the additional type rules we need for *PLinks*, compared to Lindley and Cheney's description of Links' type system [15]. The PROV rule allows us to read a value's provenance using the **prov** keyword. If M is an expression with type Prov o we can read its provenance. Provenance is returned as a record (abbreviated *PR*) with fields relation, column, and row. For the time being we represent relation and column by their names and

$$PR = \langle \textsf{relation: String, column: String, row: Int} \rangle$$

$$\textsc{Prov}\ \frac{M : \textsf{Prov}\ o}{\textbf{prov}\ M : PR} \qquad \textsc{Data}\ \frac{M : \textsf{Prov}\ o}{\textbf{data}\ M : o}$$

$$\textsc{Table}\ \frac{i \in I,\ \ p \in P,\ \ P \subseteq I \qquad o_i\ \textit{base type} \qquad f_p : \langle \overline{l_i : o_i} \rangle \to PR}{\textbf{table}\ t\ \textbf{with}\ (\overline{l_i : o_i})\ \textbf{prov}\ (\overline{l_p = f_p})\ :\ \left[\left\langle \overline{l_i : \begin{cases} \textsf{Prov}\ o_i & i \in P \\ o_i & i \notin P \end{cases}} \right\rangle \right]}$$

Figure 4. Additional typing rules compared to Links [15].

the row by an assumed numerical primary key. The **data** keyword projects a value with provenance to just the data, thus the DATA rule says that the data-part has type o if the term has type Prov o.

The TABLE rule introduces Prov types. As in Links, the programmer declares a table t with some columns l_i that contain values of base types o_i. In *PLinks*, the additional **prov** clause allows programmers to declare a function f_p for each of a subset l_p of the columns l_i. This function will be used to compute the provenance of a value of column p. It take a database row as its input and produces a provenance record (*PR*) and needs to be database-executable. The type of a **table** declaration is list of records with the declared fields. Fields l_p that appear in the **prov** clause have type Prov o_p. Fields l_i without provenance have just type o_i.

Note that all operations that would introduce "bottom" provenance in Buneman et al.'s presentation of where-provenance are ill-typed in *PLinks*. We have decided against adding a "bottom" constructor and corresponding type rule for now. It would weaken the guarantees the provenance type gives us and it seems like most, if not all, cases where it is useful can be covered by wrapping values in an explicit representation of alternatives, like Haskell's Either.

3.2 Translation

We intend to implement *PLinks*, as indicated in Figure 3, by translating *PLinks* programs into Links programs. Thus, we need to express the new keywords and types of *PLinks* in terms of Links features.

As an example of the translation, we will translate the **query** block in Figure 2. It refers to the top_comments table, filters out tuples based on their provenance using the watch_comment function, and finally returns the contents of the text column together with its provenance. We translate this piece of *PLinks* code into the Links code shown in Figure 5.

Types of the form *Prov o* are special during type checking. After that, we replace them by record types with a data and prov field. The data field has type o and will contain the actual data. The data field has the usual provenance record type and will contain provenance metadata. In the example, we see the translation of Prov o types in the signature of watch_comment. Where there used to be Prov String in Figure 2, there is a record type in Figure 5.

In the body of watch_comment we see how to translate the *PLinks* keywords **prov** and **data**. Corresponding to the translation of the Prov o type, they are simply translated into projections. Note that the less restrictive record types, compared to Prov o, do not enable programmers to circumvent the type system restrictions because we only translate programs that have passed the more restrictive type checker already.

So far, the translation was very straightforward syntactic sugar on top of records. Thus, the meat of the translation has to happen in creating these records. Indeed, we see that where we have just a reference to top_comments in Figure 2, we have a whole nested

```
sig watch_comment :
 ((data: String,
   prov: (relation: String, column: String, row: Int))) -> Bool
fun watch_comment(c) {
  c.prov.relation == "watch" || c.data =~ /.*pWatch.*/
}

query {
  for (c <-- (for (c_prime <-- top_comments)
                [(id = c_prime.id,
                  text = (data = c_prime.text,
                          prov = (fun (c) { (relation = c.origin_table,
                                             column = c.origin_column,
                                             row = c.origin_row) })
                                  (c_prime)))]))
  where (watch_comment(c.text))
    [(text = c.text)]
}
```

Figure 5. Translated query block from Figure 2.

```
SELECT
    c.text AS text_data,
    c.origin_column AS text_prov_column,
    c.origin_table AS text_prov_relation,
    c.origin_row AS text_prov_row
FROM top_comments AS c
WHERE c.origin_table = 'watch' OR c.text LIKE '%pWatch%'
```

Figure 6. SQL query generated for the code from Figures 2 and 5.

for comprehension in Figure 5. This **for** comprehension attaches provenance to the tuples of the top_comments relation. Columns whose provenance we do not care about get copied directly. In the example this is the id column and we omit code that copies the origin_* columns, as they are not used. Columns with provenance, in the example only text, are replaced by a record. The data field contains the actual value. The prov field contains the value's provenance. The provenance is computed by calling the function from the **prov** clause of the **table** declaration on each row c_prime.

Readers who are very familiar with Links might have noticed that the type of the query block is [(text: (data: String, prov: ...))], which is not legal in Links. Query blocks have to have flat relational type, that is a list of records with fields of base type, whereas here we have a list of records with fields of record type. The solution is to flatten down the record for generating the SQL query, and build only build it up after receiving the results from the database. There is a version of Links that deals with nested collections [6] that contains code to that end which we could port to mainline Links. Giorgidze et al. [11] do similar things for any nonrecursive algebraic data type. Alternatively, we could extend the translation described here to emit the necessary post-processing step instead of extending Links.

Assuming a suitable flattening, Links will do its usual query normalization and generate an SQL query similar to the one in Figure 6. The actual normalization algorithm has been described in detail elsewhere [8, 15]. A well-typed **query** block is guaranteed to result in a single SQL query. This result carries over to *PLinks* because where-provenance does not lead to nested collection types in the query result. Every "cell" is paired up with at most one triple of provenance metadata. Thus, Links will always generate a reasonable query even when the input looks somewhat convoluted like the nested **for** comprehensions, function applications, and intermediate records and projections in Figure 5.

4. Related work

To the best of our knowledge, we are the first to propose supporting provenance by translation within a programming language (leveraging language-integrated query support) rather than by altering the database system or extending it with stored procedures [1, 7, 12]. While we have focused on supporting provenance by translation to SQL, our approach also should be able to cope with database-side support for provenance, by generating queries in provenance aware query languages such as Trio's TriQL or Karvounarakis et al.'s ProQL [14].

Corcoran et al. [10] developed SELinks, a version of Links extended with label-based security enforcement, including provenance-like label propagation; this was implemented using a theory of type coercions [17]. However, their approach relied on a user defined type and stored procedures for storing and manipulating sets of labels.

Glavic et al. [13] compiled this list of requirements that a provenance system should implement: (1) support different kinds of provenance, (2) support complex queries, by which they mean a large subset of SQL if not everything, (3) complex queries over provenance data itself, and (4) scale to large databases. Although we have only considered where-provenance for the relatively simple subset of SQL supported by Links to date, we believe our approach can be extended to handle other forms of provenance and to handle richer queries. In contrast to other approaches, we have a type system that restricts operations to those that make sense in the presence of where-provenance. Our approach addresses requirement (3) for where-provenance in a new way. Regarding scalability, we have not carried out a detailed experimental evaluation, and it will be interesting to compare our approach with existing techniques based on changing the database system or extending it with stored procedures.

5. Future work

PLinks as described in Section 3 has not yet been implemented: although we have a preliminary implementation of the where-provenance translation from Buneman et al. [3], we have not yet extended Links with the Prov type constructor, the **prov** and **data** keywords, or the **prov** table modifier as described in Section 3. This is the obvious first step before gathering experience, performance evaluation, and revising the design.

Another area for future extension is database updates. When writing data to a database we can refer to the data's provenance and thus implement provenance propagation across tables.

In our current design, we make programmers drop provenance explicitly using the **data** keyword. In the future, we hope to insert it implicitly when needed. This will reduce the amount of code that needs to be adapted when adding provenance to a column.

Links has automatic type inference, but we currently require programmers to annotate functions that deal with provenance. Type inference for provenance might raise the question of polymorphism with respect to provenance.

PLinks uses programmer-defined functions to map a row to a columns provenance. This admits some flexibility in how exactly provenance is stored. It is tempting to think that we can use this to retrieve provenance for a record from another table like this:

```
table "t" with (id: Int, c : String) prov (c = fun (x) {
  THE(for (p <-- external_provenance_table)
        where (p.id == x.id)
          [(relation = p.relation,
            column = p.column,
            row = p.id)])}
```

However, this requires some function THE with type ([r]) -> r for some type record type r that is database-executable. Unfortunately, no function with this type can exist in Links as it is, because it would violate the guarantee that we create a single SQL query from a query block. At its core, this problem seems be about expressing foreign key relationships. If we were able to tell Links that some columns uniquely identify a row in another table, Links could use that information to just emit a join. How to encode such a restriction in the type system remains is not yet clear to us.

When data of type Prov o leaves a **query** block, we attach provenance metadata to it, whether it is ultimately needed or not. Where-provenance is only a constant factor in increased memory but other forms of provenance can result in larger overhead. This may turn out to be a real problem. In that case, we might want to consider lazy calculation of provenance. This would still require storing *enough* provenance to be able to query the *full* provenance. In particular, this raises the question of how to deal with a database that changes between fetching data and retrieving its provenance.

We also aim to apply the approach described here to other forms of provenance besides where-provenance, such as *why*-provenance or *how*-provenance. Prior work by Corcoran et al. [10] and Glavic et al. [12, 13] suggests that this should be possible, we intend to explore extending *PLinks* with rewriting-based support for other forms of provenance, possibly in concert with recent work on query shredding [6].

Although Links is well-suited for prototyping provenance support, it is a research language with a small audience. To make our work more accessible and useful, we plan to consider whether it is possible to apply similar ideas to (subsets of) more mainstream languages that support comprehensions or language-integrated query, such as F# or Python.

6. Conclusions

Provenance within (relational and other) databases has been investigated extensively. However, all proposals to date involve significant changes or extensions to the database system and either extend the query language (SQL) or change the query results (or both). We propose instead to support provenance at the programming language level, by translating queries so as to propagate provenance information. This strategy offers a number of possible advantages, including stronger guarantees about the integrity of provenance within the programming language, and the ability to make provenance-aware programs portable across database backends (e.g. either generating plain SQL for mainstream databases, or queries in a provenance-aware query language if the database being used supports it.) This paper reports on our preliminary implementation and proposes a refined design based on experience so far; we outline a number of possible directions for future work.

Acknowledgments

This research is supported in part by the by EU FP7 project DIACHRON (grant number 601043) and by a Google Research Award.

References

[1] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[2] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

[3] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4):28:1–28:47, December 2008.

[4] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. *Database Theory — ICDT 2001*, pages 316–330, 2001.

[5] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

[6] James Cheney, Sam Lindley, and Philip Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1027–1038, New York, NY, USA, 2014. ACM.

[7] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 942–944, 2005.

[8] Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In Philippa Gardner and Floris Geerts, editors, *Database Programming Languages*, volume 5708 of *Lecture Notes in Computer Science*, pages 36–51. Springer Berlin Heidelberg, 2009.

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 269–282, 2009.

[11] George Giorgidze, Torsten Grust, Alexander Ulrich, and Jeroen Weijers. Algebraic data types for language-integrated queries. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, DDFP '13, pages 5–10, New York, NY, USA, 2013. ACM.

[12] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 174–185, 2009.

[13] Boris Glavic, Renée J. Miller, and Gustavo Alonso. Using SQL for efficient generation and querying of provenance information. *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, pages 291–320, 2013.

[14] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 951–962, 2010.

[15] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 91–102, New York, NY, USA, 2012. ACM.

[16] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[17] Nikhil Swamy, Michael W. Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 329–340, 2009.