Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Language-integrated provenance

Stefan Fehrenbach*, James Cheney*

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

ARTICLE INFO

Article history:
Received 23 January 2017
Received in revised form 16 August 2017
Accepted 17 August 2017
Available online xxxx

Keywords:
Provenance
Language-integrated query

ABSTRACT

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

1. Introduction

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

When something goes wrong in a database-backed Web application, understanding what has gone wrong and how to fix it is also a challenge. Often, the database is the primary "state" of the program, and problems arise when this state becomes inconsistent or contains erroneous data. For example, Fig. 1 shows Links code for querying data from a (fictional) Scottish tourism database, with the result shown in Fig. 2. Suppose one of the phone numbers is incorrect: we might want to know *where* in the source database to find the source of this incorrect data, so that we can correct it. Alternatively, suppose we are curious *why* some data is produced: for example, the result shows EdinTours twice. If we were not expecting these results, e.g. because we believe that EdinTours is a bus tour agency and does not offer boat tours, then we need to see additional input data to understand why they were produced.

Automatic techniques for producing such explanations, often called *provenance*, have been explored extensively in the database literature [16,5,25]. Neither conventional nor cross-tier Web programming currently provides direct support for

* Corresponding authors.
E-mail addresses: stefan.fehrenbach@ed.ac.uk (S. Fehrenbach), jcheney@inf.ed.ac.uk (J. Cheney).

http://dx.doi.org/10.1016/j.scico.2017.08.009
0167-6423/© 2017 Elsevier B.V. All rights reserved.

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

Fig. 1. Links table declarations and example query.

Name	Phone
EdinTours	412 1200
EdinTours	412 1200
Burns's	607 3000

Fig. 2. Example query results.

provenance. A number of implementation strategies for efficiently computing provenance for query results have been explored [3,23,24], but no prior work considers the interaction of provenance with clients of the database.

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

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

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

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

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

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

Related work is discussed in greater detail in Section 7.