Language-integrated provenance in Links

Stefan Fehrenbach
James Cheney
TaPP 2015
Edinburgh, Scotland
July 9, 2015

Research supported by EU project DIACHRON and a Google Research Award
Motivation

• Lots of work on how to record, store, query provenance within a single system
  • database, WFMS, OS, ...

• Much less on how to program with that provenance
  • especially in systems spanning multiple "layers"
  • such as Web applications...
Scenario

- New, extra-nifty *pWatch* just released
- Would like to monitor comments
  - aggregated from across the Web into multiple tables
- Would like to know:
  - where did this comment come from?
  - inspect provenance to group/aggregate comments by source?
- Or maybe: delete negative comments? :)
This paper

- Initial steps towards *language-integrated provenance*

- Goals:
  - Simplify programming with provenance in web applications
  - Provide strong guarantees for "provenance safety"
    - e.g. cannot forge or (accidentally) lose provenance

- Initial focus: where-provenance for DB queries

- Building on language-integrated query (LINQ)
  - in context of the Links web/DB programming language
Basic Links program

```haskell
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.*/
}
```
Basic Links program

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.*/
}

Aggregates source data from several tables;
origin_* columns store view
or update "provenance"
Basic Links program

```
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>
}
Basic Links program

```
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>
}
```

Queries can use (some) Links functions; this will still yield a single SQL query!
Basic Links program

```
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>
}
```

Want to add a "delete this comment from source table" button...
What is Links?

- A multi-tier programming language for the Web
What is Links?

- A multi-tier programming language for the Web

Automatically partitioned to run safely on different tiers
Links overview
We describe the translation changes the types of the expressions as follows: Figure 3 shows the compilation and execution model. We type calculate their own provenance before they even reach the database. This paper extends 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 this reason, we developed an alternative approach that relies on a special "bottom" value of type

\[ \text{Prov}_{0}, \text{Prov}_{1}, \ldots, \text{Prov}_{n} \]

This does not apply to some other forms of provenance (e.g., why). The restriction of the type argument constructor indicates data paired up with its provenance. Values can not be constructed directly by a 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.

Links overview

This paper
Why Links?

- Most DB programming involves generating "query strings"
  - often dynamically
- Hence, interacting with a prov-enabled database requires pervasive changes to code and types
- In LINQ-like setting, structured query representation is available at run time already
- Hence, hope that query transformations (and associated type system changes) can be automated
Background

\[
e ::= c \mid x \mid (e_1, e_2) \mid e.i \mid e_1 + e_2 \mid e_1 = e_2 \mid \cdots \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \emptyset \mid e_1 \cup e_2 \mid \{e\} \mid \text{for } (x \leftarrow e) \text{ return } e'\n\]

\[
\tau ::= b \in \{\text{int, bool, \ldots}\} \mid t_1 \times t_2 \mid \{t\}
\]

- Nested relational calculus query expressions
- embedded in Links (LINQ similar)
Where-provenance

[Buneman, Khanna, Tan 2001]

- Where-provenance: tracks where data in output comes from

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R JOIN S</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Where-provenance
[Buneman, Khanna, Tan 2001]

- Where-provenance: tracks where data in output comes from

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 2 \\
1 & 2 & 3 \\
2 & 3 & 4 \\
\end{array}
\quad
\begin{array}{cc}
C & D \\
1 & 2 \\
2 & 2 \\
2 & 3 \\
\end{array}
\quad
\begin{array}{cccc}
A & B & C & D \\
1 & 2 & 2 & 2 \\
1 & 2 & 2 & 3 \\
\end{array}
\]
Where-provenance
[Buneman, Khanna, Tan 2001]

- Where-provenance: tracks where data in output comes from

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R JOIN S</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Where-provenance translation (simplified)
[Buneman, C., Vansummeren 2008]

\[ T(b) = b \times \text{tag} \quad T(\tau_1 \times \tau_2) = T(\tau_1) \times T(\tau_2) \quad T(\{\tau\}) = \{T(\tau)\} \]

- \[ P(x) = x \]
- \[ P(c) = (c, \perp) \]
- \[ P(e_1 \text{ op } e_2) = (P(e_1)_1 \text{ op } P(e_2)_1, \perp) \quad \text{op} \in \{+, =, \ldots\} \]
- \[ P(e.i) = P(e).i \]
- \[ P((e_1, e_2)) = (P(e_1), P(e_2)) \]
- \[ P(\text{if } e \text{ then } e_1 \text{ else } e_2) = \text{if } P(e)_1 \text{ then } P(e_1) \text{ else } P(e_2) \]
- \[ P(\emptyset) = \emptyset \]
- \[ P(e_1 \cup e_2) = P(e_1) \cup P(e_2) \]
- \[ P(\{e\}) = \{P(e)\} \]
- \[ P(\text{for } (x \leftarrow e) \text{ return } e') = \text{for } (x \leftarrow P(e)) \text{ return } P(e') \]

Key property: \( P(e) \) is flat if \( e \) is
(hence compiles to a single SQL query!)
Embedding into Links

- Added type constructor `Prov(-)`
  - `Prov t` is "a `t` with associated provenance"
  - `prov : Prov t → (relation:String,column:String,row:Int)`
  - `data : Prov t → t`
- We also allow `prov` annotations on table definitions
  - These define what data is considered "provenance" for each field
  - This can often be synthesized from existing data (e.g. keys/oids)
  - Can be different for different tables
PLinks

```plaintext
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) }
```
languages. Language-integrated query already gives uniform access to data in the programming language and data from the database. We use where-provenance, thus eliminating the need for provenance type can be certain that it carries provenance that some benefits over handling provenance manually or in the database:

- **PLinks** contains information on how to compute provenance for columns.
- **pWatch** rendered the text itself (accessed via the keyword `watch`).
- We add a new function named `delete_quote`.
- This function indicates that we expect a string argument that carries provenance.

The presence of where-provenance triple. As before, we consider comments to be about a watch, if they originate from fact, in Section 3.2 we describe how to translate into Links programs. However, language-integrated provenance has rendered this process easier.

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

sig quotes_list : () \rightarrow 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>
}
```
PLinks

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

\[ PR = \langle \text{relation: String, column: String, row: Int} \rangle \]

\[
\begin{align*}
\text{PROV} & \quad M : \text{Prov} \ o \\
\text{prov} M & : PR \\
\text{DATA} & \quad M : \text{Prov} \ o \\
\text{data} M & : o
\end{align*}
\]

<table>
<thead>
<tr>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \in I, \ p \in P, \ P \subseteq I )</td>
</tr>
<tr>
<td>( o_i ) base type</td>
</tr>
<tr>
<td>( f_p : \langle l_i : o_i \rangle \rightarrow PR )</td>
</tr>
</tbody>
</table>

\[
\text{table } t \text{ with } (l_i : o_i) \text{ prov } (l_p = f_p) : \left[ \langle l_i : \left\{ \begin{array}{ll}
\text{Prov } o_i & i \in P \\
o_i & i \notin P
\end{array} \right\} \rangle \right]
\]
Translation to plain Links

```plaintext
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)]
}
```

(this part is based on where-prov translation from [BCV08] + inlining table prov definition)
We intend to implement without provenance have just type a reference to in creating these records. Indeed, we see that where we have just on top of records. Thus, the meat of the translation has to happen checker already.

Prov o keywords has the usual provenance record type and will contain provenance that, we replace them by record types with a provenance. We translate this piece of and finally returns the contents of the block in Figure 2. It refers to the to express the new keywords and types of

3.2 Translation not all, cases where it is useful can be covered by wrapping values and finally returns the contents of the block in Figure 2. It refers to the to express the new keywords and types of

Additional typing rules compared to Links [15]. This function will be used to compute the provenance of data, they are simply translated into projections. Note that

Translated query block from Figure 2.

```sql
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%'
```

(this part relies on query translation already supported by Links)
Normalized SQL query

```sql
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%'
```

(this part relies on query translation already supported by Links)
(Desired) properties

- Type-safety (as usual)
  - added features (extra provenance "plumbing") also translate to type-safe Links code
- Provenance-safety: a value of type `Prov t` really does have "valid" provenance
  - Provenance cannot be forged!
  - No special "null" / bottom value needed for "no provenance" either
  - Provenance isn't discarded "by accident" (have to use `data` to extract raw data)
Current status / related work

- Preliminary implementation of basic translation
  - able to generate queries
  - does not execute them or return results yet
- To do next: implement Prov type, operations, and rest of translation
  - Using **data** extractor is a little painful - can we infer it?
- Longer term: consider other forms of provenance (why, how)
  - maybe using shredding to deal with set-valued annotations [C., Lindley, Wadler SIGMOD 2014]
  - or adapt other existing translations (Perm, [Alonso & Glavic 2009])
  - Also: where-provenance for updates? (cf. [Buneman, Chapman, C. 2006], [BCVo8])
Conclusions

- A typed/FP cross-tier language allows greater hope for automation, safety analysis/checking
- This is work in progress
  - but it seems like a promising way to gain experience with programming with provenance
- Of course, Links is a research prototype with O(1) users...
  - Also plan to look into transplanting ideas to other settings (e.g. LINQ in C#, F#, Scala? Python!?)