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

```
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)}</bl
  </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
}
```

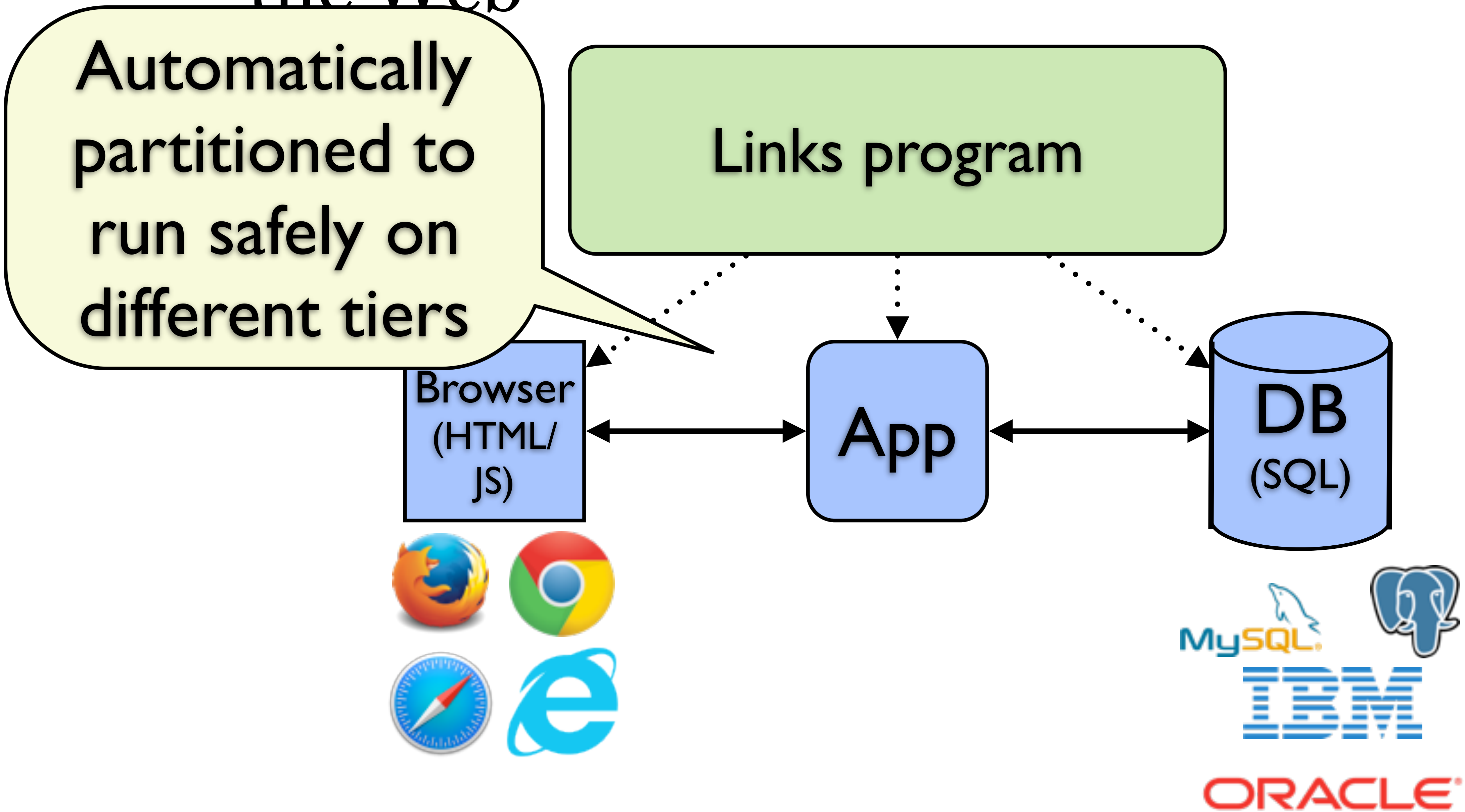
Want to add a "delete this comment from source table" button...

What is Links?

- A multi-tier programming language for the Web

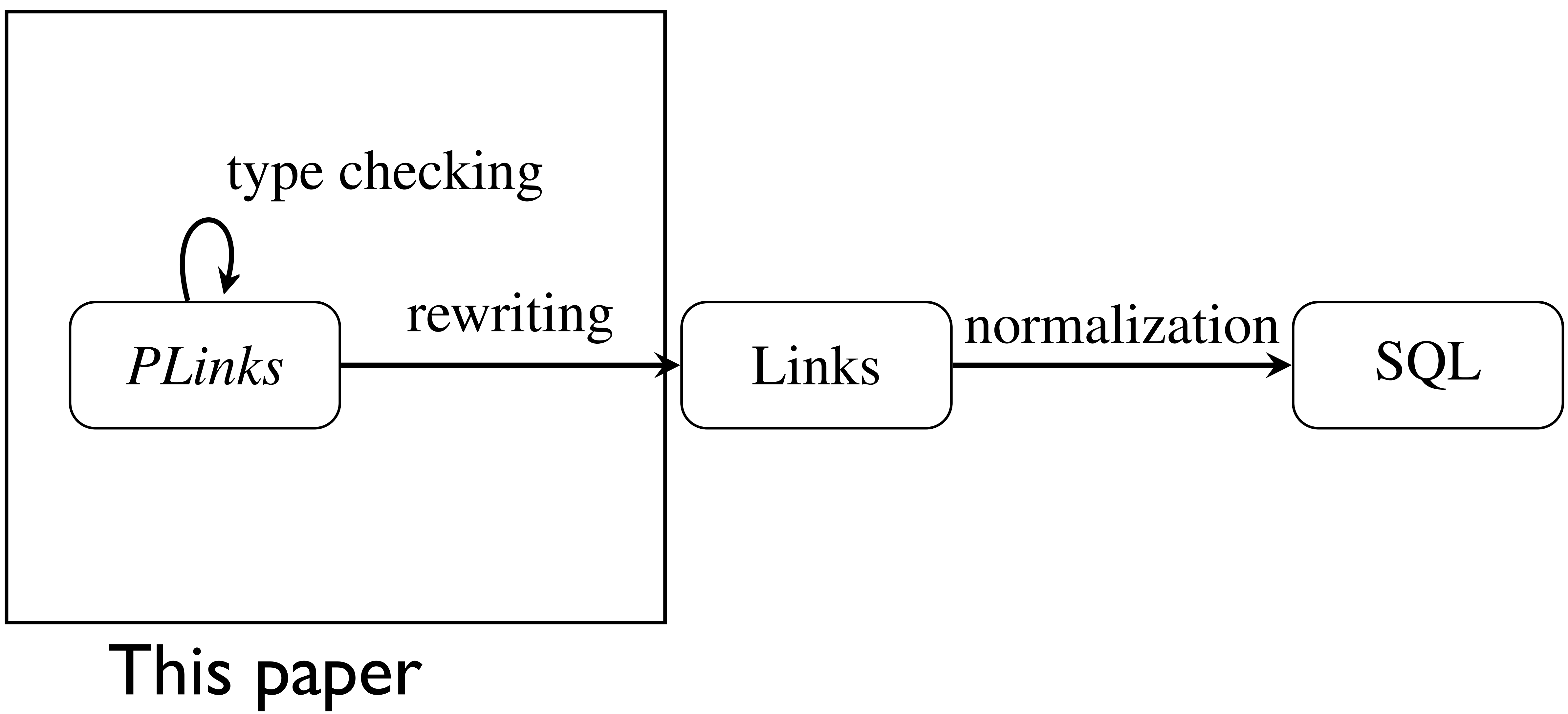
What is Links?

- A multi-tier programming language for the Web



Links overview

Links overview



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

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

- 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

R

A	B	C
1	2	2
1	2	3
2	3	4

S

C	D
1	2
2	2
2	3

R JOIN S

A	B	C	D
1	2	2	2
1	2	2	3

Where-provenance

[Buneman, Khanna, Tan 2001]

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

R

A	B	C
1	2	2
1	2	3
2	3	4

S

C	D
1	2
2	2
2	3

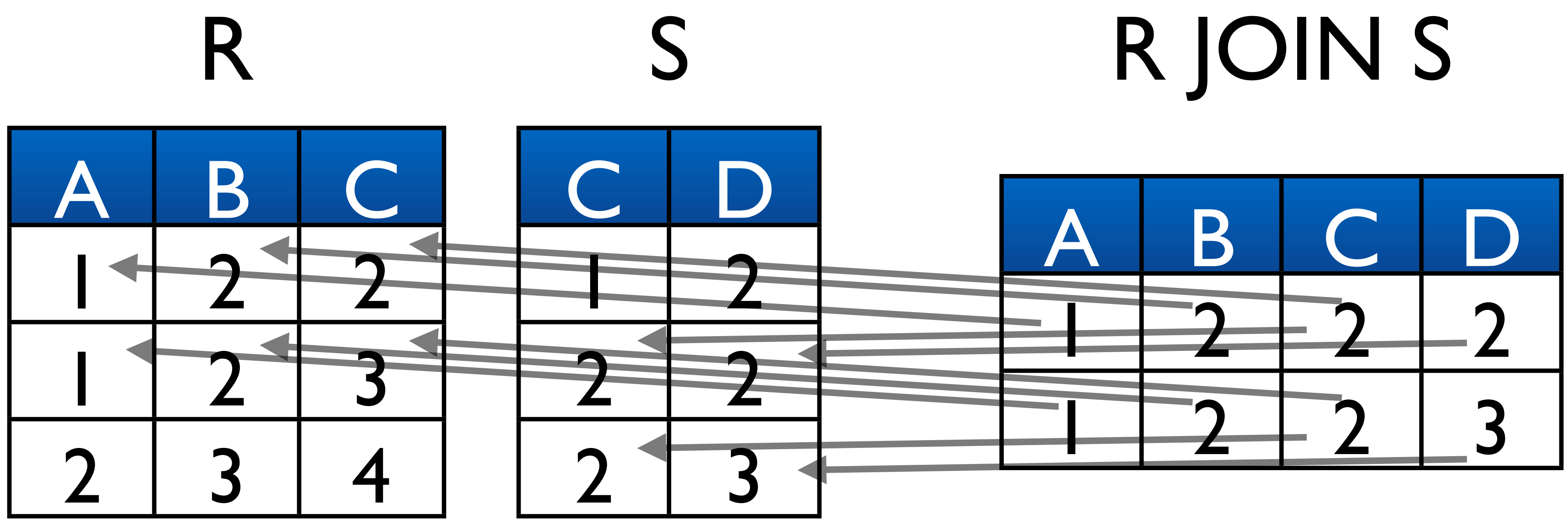
R JOIN S

A	B	C	D
1	2	2	2
1	2	2	3

Where-provenance

[Buneman, Khanna, Tan 2001]

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



Where-provenance translation (simplified)

[Buneman, C., Vansummeren 2008]

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

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

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

```
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) }
```

PLinks

```
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>
}
```

PLinks

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

Adding the delete button doesn't require changing the high-level query structure!

Types

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

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

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

Translation to plain Links

```
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)

Normalized SQL query

```
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

```
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%'
```

Sort of obvious in this case, but less so for complex queries

(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], [BCV08])

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!?)