Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

DSLDI
Prague, Tuesday 7 July 2015
How does one integrate a Domain-Specific Language and a host language?

Quotation (McCarthy, 1960)

Normalisation (Gentzen, 1935)
Part I

Getting started: Join queries
A query: Who is younger than Alex?

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>40</td>
</tr>
<tr>
<td>“Bert”</td>
<td>30</td>
</tr>
<tr>
<td>“Cora”</td>
<td>35</td>
</tr>
<tr>
<td>“Drew”</td>
<td>60</td>
</tr>
<tr>
<td>“Edna”</td>
<td>25</td>
</tr>
<tr>
<td>“Fred”</td>
<td>70</td>
</tr>
</tbody>
</table>

```sql
select v.name as name,
v.age as age
from people as u,
people as v
where u.name = “Alex” and
v.age < u.age
```

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Bert”</td>
<td>30</td>
</tr>
<tr>
<td>“Cora”</td>
<td>35</td>
</tr>
<tr>
<td>“Edna”</td>
<td>25</td>
</tr>
</tbody>
</table>
A database as data

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alex&quot;</td>
<td>40</td>
</tr>
<tr>
<td>&quot;Bert&quot;</td>
<td>30</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>35</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>60</td>
</tr>
<tr>
<td>&quot;Edna&quot;</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Fred&quot;</td>
<td>70</td>
</tr>
</tbody>
</table>

\[
\text{people = } \left[ \{ \text{name = "Alex" ; age = 40}\}; \  \{ \text{name = "Bert" ; age = 30}\}; \  \{ \text{name = "Cora" ; age = 35}\}; \  \{ \text{name = "Drew" ; age = 60}\}; \  \{ \text{name = "Edna" ; age = 25}\} ; \  \{ \text{name = "Fred" ; age = 70}\} \right]
\]
A query as F# code (naive)

```
type DB = {people : {name : string; age : int} list}
let db' : DB = database(“People”)
let youths' : {name : string; age : int} list =
    for u in db'.people do
    for v in db'.people do
    if u.name = “Alex” && v.age < u.age then
    yield {name : v.name; age : v.age}

youths' ~> 
    [ {name = “Bert” ; age = 30}
      {name = “Cora” ; age = 35}
      {name = “Edna”; age = 25} ]
```
A query as F# code (quoted)

```fsharp
type DB = {people : {name : string; age : int} list}
let db : Expr<DB> = @{database("People") @>
let youths : Expr<{name : string; age : int} list> = 
    @{for u in (%db).people do 
    for v in (%db).people do 
    if u.name = "Alex" && v.age < u.age then 
    yield {name : v.name; age : v.age} @>

run(youths) ~> 
[ {name = "Bert" ; age = 30} 
{name = "Cora" ; age = 35} 
{name = "Edna" ; age = 25} ]
```
What does \textbf{run} do?

1. Simplify quoted expression
2. Translate query to SQL
3. Execute SQL
4. Translate answer to host language

\textbf{Theorem}

Each \textbf{run} generates one query if

\textbf{A.} answer type is flat (list of record of scalars)
\textbf{B.} only permitted operations (e.g., no recursion)
\textbf{C.} only refers to one database
Scala (naive)

```scala
val youth : List[List[Record]] =
  for {u ← db.people
    v ← db.people
    if u.name == "Alex" && v.age < u.age}
  yield new Record { val name = v.name; val age = v.age }
```

Scala (quoted)

```scala
val youth : Rep[List[List[Record]]] =
  for {u ← db.people
    v ← db.people
    if u.name == "Alex" && v.age < u.age}
  yield new Record { val name = v.name; val age = v.age }
```
Part II

Nested intermediate data
### Flat data

<table>
<thead>
<tr>
<th>departments</th>
<th>employees</th>
<th>tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpt</td>
<td>dpt</td>
<td>emp</td>
</tr>
<tr>
<td>“Product”</td>
<td>“Product”</td>
<td>“Alex”</td>
</tr>
<tr>
<td>“Quality”</td>
<td>“Product”</td>
<td>“Bert”</td>
</tr>
<tr>
<td>“Research”</td>
<td>“Research”</td>
<td>“Cora”</td>
</tr>
<tr>
<td>“Sales”</td>
<td>“Sales”</td>
<td>“Drew”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Edna”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Fred”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tsk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“build”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“build”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“abstract”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“call”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“call”</td>
</tr>
</tbody>
</table>
Importing the database

```plaintext
type Org = {departments : {dpt : string} list;
            employees : {dpt : string; emp : string} list;
            tasks : {emp : string; tsk : string} list }

let org : Expr< Org > = <@ database("Org") @>
```
Departments where every employee can do a given task

\[
\begin{align*}
\text{let expertise}' & : \text{Expr< string } \rightarrow \{ \text{dpt : string} \} \text{ list } = \\
\text{<@ fun(u) } & \rightarrow \text{ for d in (%org).departments do} \\
& \text{if not(exists(} \\
& \text{ for e in (%org).employees do} \\
& \text{if d.dpt = e.dpt } \&\& \text{ not(exists(} \\
& \text{ for t in (%org).tasks do} \\
& \text{if e.emp = t.emp } \&\& \text{ t.tsk = u then yield } \{ \} \}) \\
& \text{ then yield } \{ \} \}) \\
& \text{ then yield } \{ \text{dpt = d.dpt} \} @> \\
\end{align*}
\]

\[
\begin{align*}
\text{run(<@ (%expertise')(“abstract”) @>)} \\
[ \{ \text{dpt = “Quality”} \}; \{ \text{dpt = “Research”} \} ]
\end{align*}
\]
Nested data

```javascript
[{dpt = "Product"; employees =
  [{emp = "Alex"; tasks = ["build"]}
   {emp = "Bert"; tasks = ["build"]}],
  {dpt = "Quality"; employees = []},
  {dpt = "Research"; employees =
    [{emp = "Cora"; tasks = ["abstract","build","design"]}
     {emp = "Drew"; tasks = ["abstract","design"]}
     {emp = "Edna"; tasks = ["abstract","call","design"]}],
  {dpt = "Sales"; employees =
    [{emp = "Fred"; tasks = ["call"]}]})
```
Nested data from flat data

```plaintext
type NestedOrg = [ {dpt : string; employees :
    [{emp : string; tasks : [ string ] } ] } ]

let nestedOrg : Expr< NestedOrg > =
    <@ for d in (%org).departments do
        yield {dpt = d.dpt; employees =
            for e in (%org).employees do
                if d.dpt = e.dpt then
                    yield {emp = e.emp; tasks =
                        for t in (%org).tasks do
                            if e.emp = t.emp then
                                yield t.tsk} }} @>
```
Higher-order queries

let any : Expr<(A list, A → bool) → bool> =
  <@ fun(xs, p) →
    exists(for x in xs do
      if p(x) then
        yield { } ) @>

let all : Expr<(A list, A → bool) → bool> =
  <@ fun(xs, p) →
    not((%any)(xs, fun(x) → not(p(x)))) @>

let contains : Expr<(A list, A) → bool> =
  <@ fun(xs, u) →
    (%any)(xs, fun(x) → x = u) @>
Departments where every employee can do a given task

```plaintext
let expertise : Expr< string → {dpt : string} list > =
<@ fun(u) → for d in (%nestedOrg)
  if (%all)(d.employees,
    fun(e) → (%contains)(e.tasks, u) then
    yield {dpt = d.dpt} @>

run(<@ (%expertise)(“abstract”) @>)
[ {dpt = “Quality”}; {dpt = “Research”} ]
```
Part III

Conclusion
How does one integrate a Domain-Specific Language and a host language?

Quotation (McCarthy, 1960)

Normalisation (Gentzen, 1935)
The script-writers dream, Cooper, DBPL, 2009.


Everything old is new again: Quoted Domain Specific Languages, Najd, Lindley, Svenningsson, Wadler, Draft, 2015.

Propositions as types, Wadler, CACM, to appear.


Ezra Cooper∗†, James Cheney∗, Sam Lindley*, Shayan Najd∗†, Josef Svenningsson‡, Philip Wadler*

*University of Edinburgh, †Google, ‡Chalmers University