Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

Cambridge, Thursday 11 August 2016
How does one integrate a Domain-Specific Language and a host language?

Quotation (McCarthy, 1960)

Normalisation (Gentzen, 1935)
A functional language is a domain-specific language for creating domain-specific languages
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>

```
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
```
A database as data

<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>

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

```fsharp
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 run do?

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

Theorem

Each run generates one query if
A. answer type is flat (list of record of scalars)
B. only permitted operations (e.g., no recursion)
C. only refers to one database
Scala (naive)

```scala
val youth' : List [{ val name : String; val age : Int}] =
  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 [{ val name : String; val age : Int}] ] =
  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

Abstraction, composition, dynamic generation
Abstracting over values

```plaintext
let range : Expr< (int, int) → Names > =
  @ fun(a, b) → for w in (%db).people do
    if a ≤ w.age && w.age < b then
      yield {name : w.name} @>

run(@ (%range)(30, 40) @>)

select w.name as name
from people as w
where 30 ≤ w.age and w.age < 40
```
Abstracting over a predicate

\[
\text{let } \text{satisfies : } \text{Expr< (int → bool) → Names >} = \\
@ \text{fun(p) → for w in } (\% \text{db}).\text{people do} \\
\quad \text{if p(w.age) then} \\
\quad \text{yield } \{ \text{name : w.name} \} @>
\]

\[
\text{run(@ (\%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)}
\]

\[
\text{select w.name as name} \\
\text{from people as w} \\
\text{where } 30 \leq w.\text{age and } w.\text{age} < 40
\]
Dynamically generated queries

```plaintext

<table>
<thead>
<tr>
<th>type Predicate =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above of int</td>
</tr>
<tr>
<td>Below of int</td>
</tr>
<tr>
<td>And of Predicate × Predicate</td>
</tr>
</tbody>
</table>

let rec P(t : Predicate) : Expr< int → bool > =

match t with
| Above(a) → <@ fun(x) → (%lift(a)) ≤ x @> |
| Below(a) → <@ fun(x) → x < (%lift(a)) @> |
| And(t, u) → <@ fun(x) → (%P(t))(x) && (%P(u))(x) @> |
```
Dynamically generated queries

\[
P(\text{And}(\text{Above}(30), \text{Below}(40)))
\]
\[\leadsto \langle @ \ \text{fun}(x) \rightarrow (\text{fun}(x_1) \rightarrow 30 \leq x_1)(x) \land (\text{fun}(x_2) \rightarrow x_2 < 40)(x) \rangle \]
\[\leadsto \langle @ \ \text{fun}(x) \rightarrow 30 \leq x \land x < 40 \rangle \]

\[
\text{run}(\langle @ (\%\text{satisfies})(\%P(\text{And}(\text{Above}(30), \text{Below}(40)))) \rangle)
\]

select w.name as name
from people as w
where 30 \leq w.age and w.age < 40
Part III

Closed quotation vs. open quotation
Dynamically generated queries, revisited

```plaintext
let rec P(t : Predicate) : Expr< int → bool > =
  match t with
  | Above(a) → @fun(x) → (%lift(a)) ≤ x @>
  | Below(a) → @fun(x) → x < (%lift(a)) @>
  | And(t, u) → @fun(x) → (%P(t))(x) && (%P(u))(x) @>

VS.

let rec P'(t : Predicate)(x : Expr< int >) : Expr< bool > =
  match t with
  | Above(a) → @( %lift(a)) ≤ (%x) @>
  | Below(a) → @( %x) < (%lift(a)) @>
  | And(t, u) → @( %P(t)(x)) && (%P(u)(x)) @>
```
Abstracting over a predicate, revisited

\[
\text{let } \text{satisfies : } \text{Expr}\langle \text{int } \rightarrow \text{bool} \rangle \rightarrow \text{Names} > = \\
<@ \text{fun}(p) \rightarrow \text{for } w \text{ in } (\%db).\text{people do} \\
\quad \text{if } p(w.\text{age}) \text{ then} \\
\quad \text{yield } \{\text{name : } w.\text{name}\} @> \\
\]

VS.

\[
\text{let } \text{satisfies'}(p : \text{Expr}\langle \text{int} \rangle \rightarrow \text{Expr}\langle \text{bool} \rangle) : \text{Expr}\langle \text{Names} \rangle = \\
<@ \text{for } w \text{ in } (\%db).\text{people do} \\
\quad \text{if } (\%p(<@ w.\text{age @}>)) \text{ then} \\
\quad \text{yield } \{\text{name : } w.\text{name}\} @>
\]
<table>
<thead>
<tr>
<th>QDSL</th>
<th>EDSL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Expr&lt; A → B &gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>Expr&lt; A × B &gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>Expr&lt; A + B &gt;</code></td>
<td>✓</td>
</tr>
</tbody>
</table>
closed quotations
vs.
open quotations

quotations of functions
\[(\text{Expr}^{<A \rightarrow B>})\]
vs.
functions of quotations
\[(\text{Expr}^{<A>} \rightarrow \text{Expr}^{<B>})\]
Part IV

The Subformula Principle
Gerhard Gentzen (1909–1945)
Natural Deduction — Gentzen (1935)

<table>
<thead>
<tr>
<th>&amp;–I</th>
<th>&amp;–E</th>
<th>v–I</th>
<th>v–E</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A \land B)</td>
<td>(A \land B)</td>
<td>(A \lor B)</td>
<td>([A] [B])</td>
</tr>
<tr>
<td>(A \land B)</td>
<td>(A)</td>
<td>(B)</td>
<td>(A \lor B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v–I</th>
<th>v–E</th>
<th>(\exists–I)</th>
<th>(\exists–E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\exists x , x)</td>
<td>(\forall x , x)</td>
<td>(\exists x , x)</td>
<td>([\exists x])</td>
</tr>
<tr>
<td>(\forall x , x)</td>
<td>(\exists x , x)</td>
<td>(\exists x , x)</td>
<td>(C)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\Rightarrow–I)</th>
<th>(\Rightarrow–E)</th>
<th>(\neg–I)</th>
<th>(\neg–E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([A])</td>
<td>(B)</td>
<td>(A \Rightarrow B)</td>
<td>(A)</td>
</tr>
<tr>
<td>(A \Rightarrow B)</td>
<td>(B)</td>
<td>(\neg A)</td>
<td>(A \neg A)</td>
</tr>
</tbody>
</table>
Natural Deduction

\[
\begin{align*}
[A]^x \\
\vdots \\
B \\
\hline
A \supset B \\
\end{align*}
\]

\[
\begin{align*}
&\quad A \supset B \quad A \\
&\hline
&\quad B \\
&\hline
A \supset B \\
\end{align*}
\]

\[
\begin{align*}
A & B & A \& B & A \& B \\
\hline
A \& B & & & B \\
\end{align*}
\]

\[
\begin{align*}
&\quad A \& B \\
&\hline
&A \\
\hline
&A \&-E \\
&\quad B \\
&\hline
&B \\
\end{align*}
\]

\[
\begin{align*}
&\quad A \& B \\
&\hline
&B \\
\hline
&B \&-E_0 \\
&\quad A \\
&\hline
&A \&-E_1 \\
\end{align*}
\]
Proof Normalisation

\[
\begin{align*}
[A]^x & \vdots \\
\vdots & \vdots \\
B & \vdash \neg I^x \\
A & \supset B & A & \vdash \neg E & \Rightarrow & A \\
\vdash E & \Rightarrow & B \\
\vdots & \vdots \\
A & B & \& - I \\
A & B & \& - I^0 & \Rightarrow & A \\
\end{align*}
\]
Subformula principle

Perhaps we may express the essential properties of such a normal proof by saying: it is not roundabout. No concepts enter into the proof than those contained in its final result, and their use was therefore essential to the achievement of that result.

— Gerhard Gentzen, 1935

(Subformula principle) Every formula occurring in a normal deduction in [Gentzen’s system of natural deduction] of \( A \) from \( \Gamma \) is a subformula of \( A \) or of some formula of \( \Gamma \).

— Dag Prawitz, 1965
The Curry-Howard homeomorphism
Alonzo Church (1903–1995)
Typed λ-calculus

\[
\begin{align*}
\text{[} x : A \text{]}^x \\
\vdots \\
N : B \\
\hline
\lambda x. N : A \rightarrow B \\
\end{align*}
\]

\[
\begin{align*}
L : A \rightarrow B \\
M : A \\
\hline
LM : B \\
\rightarrow \text{I}^x \\
\end{align*}
\]

\[
\begin{align*}
M : A \\
N : B \\
\hline
(M, N) : A \times B \\
\times \text{I} \\
\end{align*}
\]

\[
\begin{align*}
L : A \times B \\
\hline
\text{fst} \ L : A \\
\times \text{E}_0 \\
\end{align*}
\]

\[
\begin{align*}
L : A \times B \\
\hline
\text{snd} \ L : B \\
\times \text{E}_1 \\
\end{align*}
\]
Normalising terms

\[
\begin{align*}
[x : A]^x \\
\vdots \\
N : B & \rightarrow^x -I^x \\
\lambda x. N : A \rightarrow B & \quad M : A \\
\quad \quad (\lambda x. N) M : B & \rightarrow^E \quad \quad N\{M/x\} : B \\
\vdots \\
M : A & \quad N : B \\
\quad \quad (M, N) : A \times B & \times^I \\
\quad \quad \text{fst} (M, N) : A & \times^E_0 \quad \quad M : A
\end{align*}
\]
Normalisation

$$(\text{fun}(x) \rightarrow N) \ M \Rightarrow N[x := M]$$

$$\{\ell = M\}.\ell_i \Rightarrow M_i$$

$$\text{for } x \text{ in (yield } M) \text{ do } N \Rightarrow N[x := M]$$

$$\text{for } y \text{ in (for } x \text{ in } L \text{ do } M) \text{ do } N \Rightarrow \text{for } x \text{ in } L \text{ do (for } y \text{ in } M \text{ do } N)$$

$$\text{for } x \text{ in (if } L \text{ then } M) \text{ do } N \Rightarrow \text{if } L \text{ then (for } x \text{ in } M \text{ do } N)$$

$$\text{for } x \text{ in } [ ] \text{ do } N \Rightarrow [ ]$$

$$\text{for } x \text{ in } (L @ M) \text{ do } N \Rightarrow (\text{for } x \text{ in } L \text{ do } N) @ (\text{for } x \text{ in } M \text{ do } N)$$

$$\text{if true then } M \Rightarrow M$$

$$\text{if false then } M \Rightarrow [ ]$$
Applications of the Subformula Principle

- **Normalisation eliminates higher-order functions**  
  (SQL, Feldspar)

- **Normalisation eliminates nested intermediate data**  
  (SQL)

- **Normalisation fuses intermediate arrays**  
  (Feldspar)
Part V

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>“Research”</td>
<td>“Drew”</td>
</tr>
<tr>
<td></td>
<td>“Sales”</td>
<td>“Edna”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Fred”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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

```plaintext
let expertise' : Expr<String → {dpt : string} list> =
<@ fun(u) → for d in (%org).departments do
  if not(exists(
    for e in (%org).employees do
      if d.dpt = e.dpt && not(exists(
        for t in (%org).tasks do
          if e.emp = t.emp && t.tsk = u then yield { }
      )) then yield { }
  )) then yield {dpt = d.dpt} @>

run(<@ (%expertise')("abstract") @>)
[ {dpt = "Quality"}; {dpt = "Research"} ]
```
[ {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

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

\[
\begin{align*}
\text{let any : Expr} &\langle (A \text{ list, } A \rightarrow \text{bool}) \rightarrow \text{bool} \rangle = \\
&<@ \text{fun}(xs, p) \rightarrow \\
&\quad \exists \text{for } x \text{ in } xs \text{ do} \\
&\quad \quad \text{if } p(x) \text{ then} \\
&\quad \quad \quad \text{yield } \{ \} @> \\
\text{let all : Expr} &\langle (A \text{ list, } A \rightarrow \text{bool}) \rightarrow \text{bool} \rangle = \\
&<@ \text{fun}(xs, p) \rightarrow \\
&\quad \neg((\%\text{any})(xs, \text{fun}(x) \rightarrow \neg(p(x)))) @> \\
\text{let contains : Expr} &\langle (A \text{ list, } A) \rightarrow \text{bool} \rangle = \\
&<@ \text{fun}(xs, u) \rightarrow \\
&\quad (\%\text{any})(xs, \text{fun}(x) \rightarrow x = u) @>
\end{align*}
\]
Departments where every employee can do a given task

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

run(< fun("abstract") @>)
[ {dpt = "Quality"}; {dpt = "Research"} ]
```
Part VI

Compiling XPath to SQL
Part VII

Results
### SQL LINQ results (F#)

<table>
<thead>
<tr>
<th>Example</th>
<th>F# 2.0</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>differences</td>
<td>17.6</td>
<td>20.6</td>
<td>18.1</td>
<td>0.5</td>
</tr>
<tr>
<td>range</td>
<td>×</td>
<td>5.6</td>
<td>2.9</td>
<td>0.3</td>
</tr>
<tr>
<td>satisfies</td>
<td>2.6</td>
<td>×</td>
<td>2.9</td>
<td>0.3</td>
</tr>
<tr>
<td>P(t₀)</td>
<td>2.8</td>
<td>×</td>
<td>3.3</td>
<td>0.3</td>
</tr>
<tr>
<td>P(t₁)</td>
<td>2.7</td>
<td>×</td>
<td>3.0</td>
<td>0.3</td>
</tr>
<tr>
<td>expertise'</td>
<td>7.2</td>
<td>9.2</td>
<td>8.0</td>
<td>0.6</td>
</tr>
<tr>
<td>expertise</td>
<td>×</td>
<td>66.7^{av}</td>
<td>8.3</td>
<td>0.9</td>
</tr>
<tr>
<td>xp₀</td>
<td>×</td>
<td>8.3</td>
<td>7.9</td>
<td>1.9</td>
</tr>
<tr>
<td>xp₁</td>
<td>×</td>
<td>14.7</td>
<td>13.4</td>
<td>1.1</td>
</tr>
<tr>
<td>xp₂</td>
<td>×</td>
<td>17.9</td>
<td>20.7</td>
<td>2.2</td>
</tr>
<tr>
<td>xp₃</td>
<td>×</td>
<td>3744.9</td>
<td>3768.6</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Times in milliseconds; \(^{av}\) marks query avalanche.
# Feldspar results (Haskell)

<table>
<thead>
<tr>
<th></th>
<th>QDSL Feldspar</th>
<th>EDSL Feldspar</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
<td>Run</td>
<td>Compile</td>
</tr>
<tr>
<td>IPGray</td>
<td>16.96</td>
<td>0.01</td>
<td>15.06</td>
</tr>
<tr>
<td>IPBW</td>
<td>17.08</td>
<td>0.01</td>
<td>14.86</td>
</tr>
<tr>
<td>FFT</td>
<td>17.87</td>
<td>0.39</td>
<td>15.79</td>
</tr>
<tr>
<td>CRC</td>
<td>17.14</td>
<td>0.01</td>
<td>15.33</td>
</tr>
<tr>
<td>Window</td>
<td>17.85</td>
<td>0.02</td>
<td>15.77</td>
</tr>
</tbody>
</table>

Times in seconds; minimum time of ten runs.
Part VIII

Conclusion
‘Good artists copy, great artists steal’
– Pablo Picasso”
“‘Good artists copy, great artists steal’
– Pablo Picasso”
– Steve Jobs


<table>
<thead>
<tr>
<th>EDSL</th>
<th>QDSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>types</td>
</tr>
<tr>
<td>syntax (some)</td>
<td>syntax (all)</td>
</tr>
<tr>
<td>normalisation</td>
<td></td>
</tr>
</tbody>
</table>
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, PEPM, 2016.

Propositions as types, Wadler, CACM, Dec 2015.


Ezra Cooper∗†, James Cheney∗, Sam Lindley∗, Shayan Najd∗‡, Josef Svenningsson§, Philip Wadler∗
∗University of Edinburgh, †Qumulo, ‡Google, §Chalmers & HiQ