Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

DSLDISS
Lausanne, Monday 13 July 2015
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>&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>

```json
{people =
   [ {name = "Alex" ; age = 40};
    {name = "Bert" ; age = 30};
    {name = "Cora"; age = 35};
    {name = "Drew"; age = 60};
    {name = "Edna"; age = 25};
    {name = "Fred" ; 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

```plaintext
let satisfies : Expr< (int → bool) → Names > =
  @$ fun(p) → for w in (%db).people do
    if p(w.age) then
      yield {name : w.name} @>

run(@ (@satisfies)(fun(x) → 30 ≤ x && x < 40) @>)

select w.name as name
from people as w
where 30 ≤ w.age and w.age < 40
```
Dynamically generated queries

```ocaml
type Predicate =
  | Above of int
  | Below of int
  | And of Predicate × Predicate

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

$$P(\text{And}(\text{Above}(30), \text{Below}(40)))$$

$$\leadsto \text{fun}(x) \rightarrow (\text{fun}(x_1) \rightarrow 30 \leq x_1)(x) \&\& (\text{fun}(x_2) \rightarrow x_2 < 40)(x) @>$$

$$\leadsto \text{fun}(x) \rightarrow 30 \leq x \&\& x < 40 @>$$

run($$\text{fun}(x) \rightarrow (\%\text{satisfies})(\%P(\text{And}(\text{Above}(30), \text{Below}(40))))$$)

```
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

```ocaml
define (P : 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.

define (P' : predicate) -> 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

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

\text{vs.}

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

Expr< A → B >   ✓   Expr< A > → Expr< B >   ✓

Expr< A × B >   ✓   Expr< A > × Expr< B >   ✓

Expr< A + B >   ✓   Expr< A > + Expr< B >   ✗
closed quotations

vs.

open quotations

quotations of functions

\((\text{Expr}\langle A \rightarrow B \rangle)\)

vs.

functions of quotations

\((\text{Expr}\langle A \rangle \rightarrow \text{Expr}\langle B \rangle)\)
Part IV

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

\[
\begin{align*}
& \&-I \\
& \text{ \ \ \ } \frac{A \land B}{A \land B} \\

& \&-E \\
& \text{ \ \ \ } \frac{A \land B}{A} \quad \frac{A \land B}{B} \\

& \lor- I \\
& \text{ \ \ \ } \frac{A \lor B}{A} \quad \frac{A \lor B}{B} \\

& \lor- E \\
& \text{ \ \ \ } \frac{[A] \quad [B]}{A \lor B \land C \land C} \\

& \forall- I \\
& \text{ \ \ \ } \frac{[a]}{\forall x \, Fx} \\

& \forall- E \\
& \text{ \ \ \ } \frac{\forall x \, Fx}{F(a)} \\

& \exists- I \\
& \exists- E \\
& \text{ \ \ \ } \frac{F(a)}{\exists x \, Fx} \quad \frac{\exists x \, Fx}{C} \\

& \Rightarrow- I \\
& \text{ \ \ \ } \frac{[A]}{A \Rightarrow B} \\

& \Rightarrow- E \\
& \text{ \ \ \ } \frac{A \Rightarrow B}{A \lor B} \quad \frac{A \Rightarrow B}{B} \\

& \neg- I \\
& \text{ \ \ \ } \frac{[A]}{\neg A} \\

& \neg- E \\
& \text{ \ \ \ } \frac{A \neg A \land \neg A}{A} \quad \frac{A \neg A \land \neg A}{\bot} 
\end{align*}
\]
Natural Deduction

\[
\frac{\begin{array}{c}
  [A]^x \\
  \vdots \\
  B
\end{array}}{A \supset B} \supset\text{I}^x
\]

\[
\frac{A \supset B \quad A}{B} \supset\text{E}
\]

\[
\frac{A \quad B}{A \& B} \&\text{I}
\]

\[
\frac{A \& B}{A} \&\text{E}_0
\]

\[
\frac{A \& B}{B} \&\text{E}_1
\]
Proof Normalisation

\[ [A]^x \]

\[ \vdots \]

\[ B \]

\[ A \sqsupset B \]

\[ \supset \text{-}I^x \]

\[ A \]

\[ \vdash \text{-} \text{E} \]

\[ \Rightarrow \]

\[ B \]

\[ \vdots \]

\[ \vdots \]

\[ A \quad B \]

\[ \& \text{-} \text{I} \]

\[ A \& B \]

\[ \& \text{-} \text{E}_0 \]

\[ \Rightarrow \]

\[ A \]
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
Typed λ-calculus

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

\[
\frac{L : A \rightarrow B \quad M : A}{LM : B} \rightarrow^x -E
\]

\[
\begin{align*}
M & : A \\
N & : B \\
(M, N) & : A \times B \\
L & : A \times B \\
\textsf{fst} L & : A \\
\textsf{snd} L & : B \\
\end{align*}
\]

\[
\frac{\times^- I}{(M, N) : A \times B} \\
\frac{\times^- E_0}{\textsf{fst} L : A} \\
\frac{\times^- E_1}{\textsf{snd} L : B}
\]
Normalising terms

\[
\begin{align*}
&[x : A]^x \\
&\quad \vdots \\
&\quad N : B \\
&\quad \lambda x. N : A \to B \\
&\quad (\lambda x. N)M : B \\
&\quad \vdots \\
&\quad M : A
\end{align*}
\]

\[
\begin{align*}
&\quad \rightarrow^{I^x} \\
&\quad M : A \\
&\quad (\lambda x. N)M : B \\
&\quad \vdots \\
&\quad N \{M/x\} : B
\end{align*}
\]

\[
\begin{align*}
&\quad \vdots \\
&\quad M : A \\
&\quad N : B \\
&\quad (M, N) : A \times B \\
&\quad \text{fst} (M, N) : A \\
&\quad \vdots \\
&\quad M : A
\end{align*}
\]

\[
\begin{align*}
&\quad \times-I \\
&\quad \times-E_0 \\
&\quad \Rightarrow 
\end{align*}
\]
Normalisation

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

\[ \{\ell = M\}.\ell_i \leadsto M_i \]

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

\[ \text{for } y \text{ in } (\text{for } x \text{ in } L \text{ do } M) \text{ do } N \leadsto \text{for } x \text{ in } L \text{ do } (\text{for } y \text{ in } M \text{ do } N) \]

\[ \text{for } x \text{ in } (\text{if } L \text{ then } M) \text{ do } N \leadsto \text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N) \]

\[ \text{for } x \text{ in } [ ] \text{ do } N \leadsto [ ] \]

\[ \text{for } x \text{ in } (L @ M) \text{ do } N \leadsto (\text{for } x \text{ in } L \text{ do } N) @ (\text{for } x \text{ in } M \text{ do } N) \]

\[ \text{if true then } M \leadsto M \]

\[ \text{if false then } M \leadsto [ ] \]
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
<table>
<thead>
<tr>
<th>departments</th>
<th>employees</th>
<th>tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpt</td>
<td>emp</td>
<td>tsk</td>
</tr>
<tr>
<td>“Product”</td>
<td>“Alex”</td>
<td>“build”</td>
</tr>
<tr>
<td>“Quality”</td>
<td>“Bert”</td>
<td>“build”</td>
</tr>
<tr>
<td>“Research”</td>
<td>“Cora”</td>
<td>“abstract”</td>
</tr>
<tr>
<td>“Sales”</td>
<td>“Drew”</td>
<td>“design”</td>
</tr>
<tr>
<td></td>
<td>“Edna”</td>
<td>“abstract”</td>
</tr>
<tr>
<td></td>
<td>“Fred”</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

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”} ]
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

```haskell
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 VI

Compiling XQuery 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&lt;sup&gt;av&lt;/sup&gt;</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; <sup>av</sup> 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><strong>IPGray</strong></td>
<td>16.96</td>
<td>0.01</td>
<td>15.06</td>
</tr>
<tr>
<td><strong>IPBW</strong></td>
<td>17.08</td>
<td>0.01</td>
<td>14.86</td>
</tr>
<tr>
<td><strong>FFT</strong></td>
<td>17.87</td>
<td>0.39</td>
<td>15.79</td>
</tr>
<tr>
<td><strong>CRC</strong></td>
<td>17.14</td>
<td>0.01</td>
<td>15.33</td>
</tr>
<tr>
<td><strong>Window</strong></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
“‘Good artists copy, great artists steal’
   – Pablo Picasso”
   – Steve Jobs

EDSL
   types
   syntax (some)

QDSL
   types
   syntax (all)
   normalisation
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