Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

Google X, Thursday 16 November 2017
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>

{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. Normalise 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

\[
\text{let } \text{range} : \text{Expr}<(\text{int, int}) \rightarrow \text{Names}> = \\
<\triangle \text{fun}(a, b) \rightarrow \text{for } w \in (\%\text{db}).\text{people do} \\
\text{if } a \leq w.\text{age} \land w.\text{age} < b \text{ then} \\
\text{yield } \{\text{name} : w.\text{name}\} @> \\
\]

\[
\text{run}(<\triangle (\%\text{range})(30, 40) @>) \\
\]

\[
\text{select } w.\text{name as name} \\
\text{from } \text{people as } w \\
\text{where } 30 \leq w.\text{age and } w.\text{age} < 40
\]
Abstracting over a predicate

```
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

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

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

\begin{verbatim}
select w.name as name
from people as w
where 30 \leq w.age and w.age < 40
\end{verbatim}
Part III

Closed quotation vs. open quotation
Dynamically generated queries, revisited

```ocaml
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

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

\[\text{vs.}
\]

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

\[
\text{Expr} < A \to B > \quad \checkmark \quad \text{Expr} < A > \to \text{Expr} < B > \quad \checkmark
\]

\[
\text{Expr} < A \times B > \quad \checkmark \quad \text{Expr} < A > \times \text{Expr} < B > \quad \checkmark
\]

\[
\text{Expr} < A + B > \quad \checkmark \quad \text{Expr} < A > + \text{Expr} < B > \quad \text{X}
\]

EDSL
closed quotations
vs.
open quotations

quotations of functions
(Expr< A \rightarrow B >)
vs.
functions of quotations
(Expr< A > \rightarrow Expr< B >)
Part IV

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

\[\begin{array}{cccc}
\text{&–I} & \text{&–E} & \text{\lor–I} & \text{\lor–E} \\
A & B & A & B & A & B & A & B & A \lor B & A \lor B & \text{[A]} & \text{[B]} & C & C
\end{array}\]

\[\begin{array}{cccc}
\text{\lor–I} & \text{\lor–E} & \exists–I & \exists–E \\
\exists x \exists y & \exists x \exists y & \exists x & \exists x & \text{[Sx]} & \exists x \exists y & C & C
\end{array}\]

\[\begin{array}{cccc}
\text{\rightarrow–I} & \text{\rightarrow–E} & \neg–I & \neg–E \\
\text{[A]} & A & A \rightarrow B & \text{[A]} & A & A \rightarrow B & A \rightarrow A & A \rightarrow B
\end{array}\]

\[\begin{array}{cccc}
\text{\land–I} & \text{\land–E} & \land–I & \land–E \\
\land & \land & \text{[A]} & \text{[A]} & A & A & A
\end{array}\]
Natural Deduction

\[ [A]^x \]
\[ \vdash B \]
\[ \vdash A \supset B \]
\[ \vdash B \]
\[ \vdash A \supset B \]
\[ \vdash A \]
\[ \vdash A \land B \]

\[ \vdash \neg I^x \]
\[ \vdash \neg E \]

\[ \vdash \neg I \]
\[ \vdash \neg E_0 \]
\[ \vdash \neg E_1 \]
Proof Normalisation

\[
\begin{array}{c}
[A]^x \\
\vdots \\
B \\
\frac{A \supset B}{\supset-I^x} \\
\frac{A \supset B}{B} \\
\frac{\vdash-E}{\supset-E} \quad \Rightarrow \\
\vdots \\
\vdots \\
A & B \\
\frac{A \& B}{\&-I} \\
\frac{A \& B}{A} \\
\frac{\vdash-E_0}{\&-E_0} \quad \Rightarrow \\
\vdots \\
\vdots \\
\end{array}
\]
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 $\lambda$-calculus

$$\begin{align*}
[x : A]^x & \\
\vdots & \\
N : B & \\
\lambda x. N : A \to B & \Rightarrow \text{I}^x
\end{align*}$$

$$\begin{align*}
L : A \to B & \\
M : A & \\
LM : B & \Rightarrow \text{E}
\end{align*}$$

$$\begin{align*}
M : A & \\
N : B & \\
(M, N) : A \times B & \Rightarrow \text{I}
\end{align*}$$

$$\begin{align*}
L : A \times B & \\
\times-E_0 & \\
\text{fst} L : A & \\
\text{snd} L : B & \Rightarrow \text{E}_1
\end{align*}$$
Normalising terms

\[
[x : A]^x \\
\vdots \\
N : B \quad \rightarrow^x \quad -I^x \\
\lambda x. N : A \rightarrow B \\
\vdots \\
(\lambda x. N) M : B \quad \rightarrow^E \quad \Longrightarrow \\
\vdots \\
M : A
\]

\[
\vdots \\
\vdots \\
M : A \\
N : B \\
(\lambda x. N) M : B \\
\vdots \\
(\lambda x. N) M : B \\
\vdots \\
M : A
\]
Normalisation

\[
\begin{align*}
\text{(fun}(x) \to N) \ M & \leadsto N[x := M] \\
\{\ell = M\}.\ell_i & \leadsto M_i \\
\text{for } x \text{ in (yield } M \text{) do } N & \leadsto N[x := M] \\
\text{for } y \text{ in (for } x \text{ in } L \text{ do } M \text{) do } N & \leadsto \text{for } x \text{ in } L \text{ do (for } y \text{ in } M \text{ do } N) \\
\text{for } x \text{ in (if } L \text{ then } M \text{) do } N & \leadsto \text{if } L \text{ then (for } x \text{ in } M \text{ do } N) \\
\text{for } x \text{ in } [ ] \text{ do } N & \leadsto [ ] \\
\text{for } x \text{ in (} L \text{ @} M \text{) do } N & \leadsto (\text{for } x \text{ in } L \text{ do } N) \text{ @} (\text{for } x \text{ in } M \text{ do } N) \\
\text{if true then } M & \leadsto M \\
\text{if false then } M & \leadsto [ ]
\end{align*}
\]
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>
</tbody>
</table>

**Tables:**
- **departments**: “Product”, “Quality”, “Research”, “Sales”
- **tasks**: “build”, “abstract”, “design”, “call”
Departments where every employee can abstract

```
select d.dpt as dpt
from departments as d
where not(exists(
  select *
  from employees as e
  where d.dpt = e.dpt and not(exists(
    select *
    from tasks as t
    where e.emp = t.emp and t.tsk = "abstract"
  ))
))
```
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′ : $\text{Expr< \ string \ → \ \{dpt : \ string\} \ list}$ =
<@ 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 
)) then yield 

run(<@ (%expertise′)(“abstract”) @>)
[ 
  \{dpt = “Quality”\}; \{dpt = “Research”\}]
Nested data

```json
[ {
  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

\[
\text{let } \text{any} : \text{Expr}< (A \ \text{list}, \ A \to \text{bool}) \rightarrow \text{bool} > = \\
<@ \text{fun}(xs, p) \rightarrow \\
\quad \text{exists}(\text{for } x \ \text{in } xs \ \text{do} \\
\quad \quad \text{if } p(x) \ \text{then} \\
\quad \quad \quad \text{yield } \{ \} ) @> \\
\text{let } \text{all} : \text{Expr}< (A \ \text{list}, \ A \to \text{bool}) \rightarrow \text{bool} > = \\
<@ \text{fun}(xs, p) \rightarrow \\
\quad \text{not}(\text{not}(\text{any}(xs, \text{fun}(x) \rightarrow \text{not}(p(x)))))) @> \\
\text{let } \text{contains} : \text{Expr}< (A \ \text{list}, \ A) \rightarrow \text{bool} > = \\
<@ \text{fun}(xs, u) \rightarrow \\
\quad (\text{any}(xs, \text{fun}(x) \rightarrow x = u)) @>
\]
Departments where every employee can do a given task

\[
\text{let} \ \text{expertise} : \text{Expr< string } \rightarrow \{ \text{dpt : string}\} \ \text{list} > = \n\text{@ fun(u) } \rightarrow \text{for d in } \%\text{nestedOrg} \n\quad \text{if } \%\text{all}(d.employees,} \n\quad \quad \text{fun(e) } \rightarrow \%\text{contains}(e.tasks, u) \ \text{then} \n\quad \text{yield } \{ \text{dpt = d.dpt} \} \quad \text{@} \n\text{run(}@ \%\text{expertise}(\text{“abstract”}) @> \text{)} \n\text{[ }\{ \text{dpt = “Quality”}\}; \{ \text{dpt = “Research”}\} \text{]}
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></td>
<td></td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>IPBW</td>
<td>17.08</td>
<td>0.01</td>
<td>14.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>FFT</td>
<td>17.87</td>
<td>0.39</td>
<td>15.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.07</td>
</tr>
<tr>
<td>CRC</td>
<td>17.14</td>
<td>0.01</td>
<td>15.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.05</td>
</tr>
<tr>
<td>Window</td>
<td>17.85</td>
<td>0.02</td>
<td>15.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.06</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

<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></td>
<td>normalisation</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