Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler, University of Edinburgh
with Sam Lindley, Shayan Najd, Josef Svenningsson

Chalmers
Monday 25 April 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>people</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>age</td>
</tr>
<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. 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

```
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

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

\[
\leadsto \langle @ \ \text{fun}(x) \rightarrow (\text{fun}(x_1) \rightarrow 30 \leq x_1)(x) \ \&\& \ (\text{fun}(x_2) \rightarrow x_2 < 40)(x) \rangle
\]

\[
\leadsto \langle @ \ \text{fun}(x) \rightarrow 30 \leq x \ \&\& \ x < 40 \rangle
\]

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

\[
\text{select} \ w.\text{name} \text{ as name}
\]

\[
\text{from} \ \text{people} \text{ as w}
\]

\[
\text{where} \ 30 \leq w.\text{age} \ \text{and} \ w.\text{age} < 40
\]
Part III

Closed quotation vs. open quotation
Dynamically generated queries, revisited

let rec $P(t : \text{Predicate}) : \text{Expr< int } \rightarrow \text{ bool >} =$

match $t$ with

| Above(a) $\rightarrow <@ \text{fun}(x) \rightarrow ($%lift(a)) $\leq$ x $@$>
| Below(a) $\rightarrow <@ \text{fun}(x) \rightarrow x < ($%lift(a)) $@$>
| And(t, u) $\rightarrow <@ \text{fun}(x) \rightarrow ($%P(t))(x) $\&\&$ ($%P(u))(x) $@$>

VS.

let rec $P'(t : \text{Predicate})(x : \text{Expr< int >}) : \text{Expr< bool >} =$

match $t$ with

| Above(a) $\rightarrow <@ ($%lift(a)) $\leq$ ($%x) $@$>
| Below(a) $\rightarrow <@ ($%x) $<$ ($%lift(a)) $@$>
| And(t, u) $\rightarrow <@ ($%P'(t)(x)) $\&\&$ ($%P'(u)(x)) $@$>
Abstracting over a predicate, revisited

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

vs.

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

QDSL

Expr\langle A \to B \rangle

Expr\langle A \times B \rangle

Expr\langle A + B \rangle

EDSL

Expr\langle A \rangle \to Expr\langle B \rangle

Expr\langle A \rangle \times Expr\langle B \rangle

Expr\langle A \rangle + Expr\langle B \rangle
closed quotations
  vs.
open quotations

quotations of functions
  \((\text{Expr}<A \to B>)\)
  vs.
functions of quotations
  \((\text{Expr}<A> \to \text{Expr}<B>)\)
Part IV

The Subformula Principle
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
Normalisation

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

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

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

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

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

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

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

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

\[\text{if false then } M \rightsquigarrow [ ]\]
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

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>66.7av</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  Run</td>
<td>Compile  Run</td>
<td>Compile  Run</td>
</tr>
<tr>
<td>IPGray</td>
<td>16.96 0.01</td>
<td>15.06 0.01</td>
<td>0.06 0.39</td>
</tr>
<tr>
<td>IPBW</td>
<td>17.08 0.01</td>
<td>14.86 0.01</td>
<td>0.06 0.19</td>
</tr>
<tr>
<td>FFT</td>
<td>17.87 0.39</td>
<td>15.79 0.09</td>
<td>0.07 3.02</td>
</tr>
<tr>
<td>CRC</td>
<td>17.14 0.01</td>
<td>15.33 0.01</td>
<td>0.05 0.12</td>
</tr>
<tr>
<td>Window</td>
<td>17.85 0.02</td>
<td>15.77 0.01</td>
<td>0.06 0.27</td>
</tr>
</tbody>
</table>

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

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                       QDSL

  types                       types
  syntax (some)               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.
A practical theory of language integrated query,
Everything old is new again: Quoted Domain Specific Languages,
Propositions as types, Wadler, CACM, December 2015.


Ezra Cooper*,†, James Cheney*, Sam Lindley*,
Shayan Najd*,†, Josef Svenningsson‡, Philip Wadler*
*University of Edinburgh, †Google, ‡Chalmers University