Everything old is new again:
Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

IBM, Watson
Monday 2 May 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>&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>

\[
\text{people} = \\
\{ \{ \text{name} = "Alex" ; \text{age} = 40 \} ; \\
\{ \text{name} = "Bert" ; \text{age} = 30 \} ; \\
\{ \text{name} = "Cora" ; \text{age} = 35 \} ; \\
\{ \text{name} = "Drew" ; \text{age} = 60 \} ; \\
\{ \text{name} = "Edna" ; \text{age} = 25 \} ; \\
\{ \text{name} = "Fred" ; \text{age} = 70 \} \} 
\]
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

let range : \texttt{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

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

\[
\text{run}(<@ (\% \text{satisfies})(\text{fun}(x) \rightarrow 30 \leq x \land x < 40) @>) \\
\]

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

```

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

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

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

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

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

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

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

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

Closed quotation vs. open quotation
Dynamically generated queries, revisited

\[
\text{let rec } P(t : \text{Predicate}) : \text{Expr< int } \rightarrow \text{ bool }> = \\
\text{match } t \text{ with} \\
| \text{Above}(a) \rightarrow <@ \text{fun}(x) \rightarrow (\%\text{lift}(a)) \leq x @> \\
| \text{Below}(a) \rightarrow <@ \text{fun}(x) \rightarrow x < (\%\text{lift}(a)) @> \\
| \text{And}(t, u) \rightarrow <@ \text{fun}(x) \rightarrow (\%P(t))(x) \&\& (\%P(u))(x) @>
\]

\text{VS.}

\[
\text{let rec } P'(t : \text{Predicate})(x : \text{Expr< int >}) : \text{Expr< bool >} = \\
\text{match } t \text{ with} \\
| \text{Above}(a) \rightarrow <@ (\%\text{lift}(a)) \leq (\%x) @> \\
| \text{Below}(a) \rightarrow <@ (\%x) < (\%\text{lift}(a)) @> \\
| \text{And}(t, u) \rightarrow <@ (\%P'(t)(x)) \&\& (\%P'(u)(x)) @>
\]
Abstracting over a predicate, revisited

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

\text{vs.}

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

quotations of functions
(\textit{Expr}< A \rightarrow B >)
vs.
functions of quotations
(\textit{Expr}< A > \rightarrow \textit{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

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

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

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

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

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

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

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

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

\[
\text{if false then } M \rightsquigarrow [ ]
\]
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(t_0))</td>
<td>2.8</td>
<td>×</td>
<td>3.3</td>
<td>0.3</td>
</tr>
<tr>
<td>(P(t_1))</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>xp(_0)</td>
<td>×</td>
<td>8.3</td>
<td>7.9</td>
<td>1.9</td>
</tr>
<tr>
<td>xp(_1)</td>
<td>×</td>
<td>14.7</td>
<td>13.4</td>
<td>1.1</td>
</tr>
<tr>
<td>xp(_2)</td>
<td>×</td>
<td>17.9</td>
<td>20.7</td>
<td>2.2</td>
</tr>
<tr>
<td>xp(_3)</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 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.


Everything old is new again: Quoted Domain Specific Languages, Najd, Lindley, Svenningsson, Wadler, PEPM, 2016.

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