

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?

people

name	age
"Alex"	40
"Bert"	30
"Cora"	35
"Drew"	60
"Edna"	25
"Fred"	70

```
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
```

answer

name	age
"Bert"	30
"Cora"	35
"Edna"	25

A database as data

people

name	age
"Alex"	40
"Bert"	30
"Cora"	35
"Drew"	60
"Edna"	25
"Fred"	70

{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)

```
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' $\rightsquigarrow$

```
[ {name = "Bert" ; age = 30}  
  {name = "Cora"; age = 35}  
  {name = "Edna"; age = 25} ]
```

A query as F# code (quoted)

```
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) $\rightsquigarrow$

```
[ {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)

```
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)

```
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 : 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 $\rightarrow$ bool) $\rightarrow$ Names > =

<@ **fun**(p) $\rightarrow$ **for** w **in** (%db).people **do**

if p(w.age) **then**

yield {name : w.name} @>

run(<@ (%satisfies)(**fun**(x) $\rightarrow$ $30 \leq x \ \&\& \ x < 40$) @>)

select w.name **as** name

from people **as** w

where $30 \leq$ w.age **and** w.age < 40

Dynamically generated queries

```
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

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

$\rightsquigarrow \langle @ \text{ fun}(x) \rightarrow (\text{fun}(x_1) \rightarrow 30 \leq x_1)(x) \ \&\& \ (\text{fun}(x_2) \rightarrow x_2 < 40)(x) \ @ \rangle$

$\rightsquigarrow \langle @ \text{ fun}(x) \rightarrow 30 \leq x \ \&\& \ x < 40 \ @ \rangle$

`run(<@ (%satisfies)(%P(And(Above(30), Below(40)))) @>)`

`select w.name as name`

`from people as w`

`where 30 ≤ w.age and w.age < 40`

Part III

Closed quotation vs. open quotation

Dynamically generated queries, revisited

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

```
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

$\text{Expr} \langle A \rightarrow B \rangle$ ✓

$\text{Expr} \langle A \times B \rangle$ ✓

$\text{Expr} \langle A + B \rangle$ ✓

EDSL

$\text{Expr} \langle A \rangle \rightarrow \text{Expr} \langle B \rangle$ ✓

$\text{Expr} \langle A \rangle \times \text{Expr} \langle B \rangle$ ✓

$\text{Expr} \langle A \rangle + \text{Expr} \langle B \rangle$ ✗

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)

$\&-I$ $\frac{\mathcal{A} \quad \mathcal{B}}{\mathcal{A} \& \mathcal{B}}$	$\&-E$ $\frac{\mathcal{A} \& \mathcal{B}}{\mathcal{A}} \quad \frac{\mathcal{A} \& \mathcal{B}}{\mathcal{B}}$	$\vee-I$ $\frac{\mathcal{A}}{\mathcal{A} \vee \mathcal{B}} \quad \frac{\mathcal{B}}{\mathcal{A} \vee \mathcal{B}}$	$\vee-E$ $\frac{\mathcal{A} \vee \mathcal{B} \quad \begin{array}{c} [\mathcal{A}] \\ \mathcal{C} \end{array} \quad \begin{array}{c} [\mathcal{B}] \\ \mathcal{C} \end{array}}{\mathcal{C}}$
$\forall-I$ $\frac{\mathcal{F}a}{\forall x \mathcal{F}x}$	$\forall-E$ $\frac{\forall x \mathcal{F}x}{\mathcal{F}a}$	$\exists-I$ $\frac{\mathcal{F}a}{\exists x \mathcal{F}x}$	$\exists-E$ $\frac{\exists x \mathcal{F}x \quad \begin{array}{c} [\mathcal{F}a] \\ \mathcal{C} \end{array}}{\mathcal{C}}$
$\supset-I$ $\frac{\begin{array}{c} [\mathcal{A}] \\ \mathcal{B} \end{array}}{\mathcal{A} \supset \mathcal{B}}$	$\supset-E$ $\frac{\mathcal{A} \quad \mathcal{A} \supset \mathcal{B}}{\mathcal{B}}$	$\neg\neg-I$ $\frac{\begin{array}{c} [\mathcal{A}] \\ \wedge \end{array}}{\neg \mathcal{A}}$	$\neg\neg-E$ $\frac{\mathcal{A} \neg \mathcal{A} \quad \frac{\wedge}{\mathcal{D}}}{\wedge} .$

Natural Deduction

$$\frac{\begin{array}{c} [A]^x \\ \vdots \\ B \end{array}}{A \supset B} \supset\text{-I}^x \qquad \frac{A \supset B \quad A}{B} \supset\text{-E}$$

$$\frac{A \quad B}{A \& B} \&\text{-I} \qquad \frac{A \& B}{A} \&\text{-E}_0 \qquad \frac{A \& B}{B} \&\text{-E}_1$$

Proof Normalisation

$$\begin{array}{c}
 [A]^x \\
 \vdots \\
 B \\
 \hline
 A \supset B \quad \supset\text{-I}^x
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 A \\
 \supset\text{-E}
 \end{array}
 \quad
 \frac{}{B}
 \quad
 \Rightarrow
 \quad
 \begin{array}{c}
 \vdots \\
 A \\
 \vdots \\
 B
 \end{array}$$

$$\begin{array}{c}
 \vdots \quad \vdots \\
 A \quad B \\
 \hline
 A \& B \quad \&\text{-I} \\
 \hline
 A \quad \&\text{-E}_0
 \end{array}
 \quad
 \Rightarrow
 \quad
 \begin{array}{c}
 \vdots \\
 A
 \end{array}$$

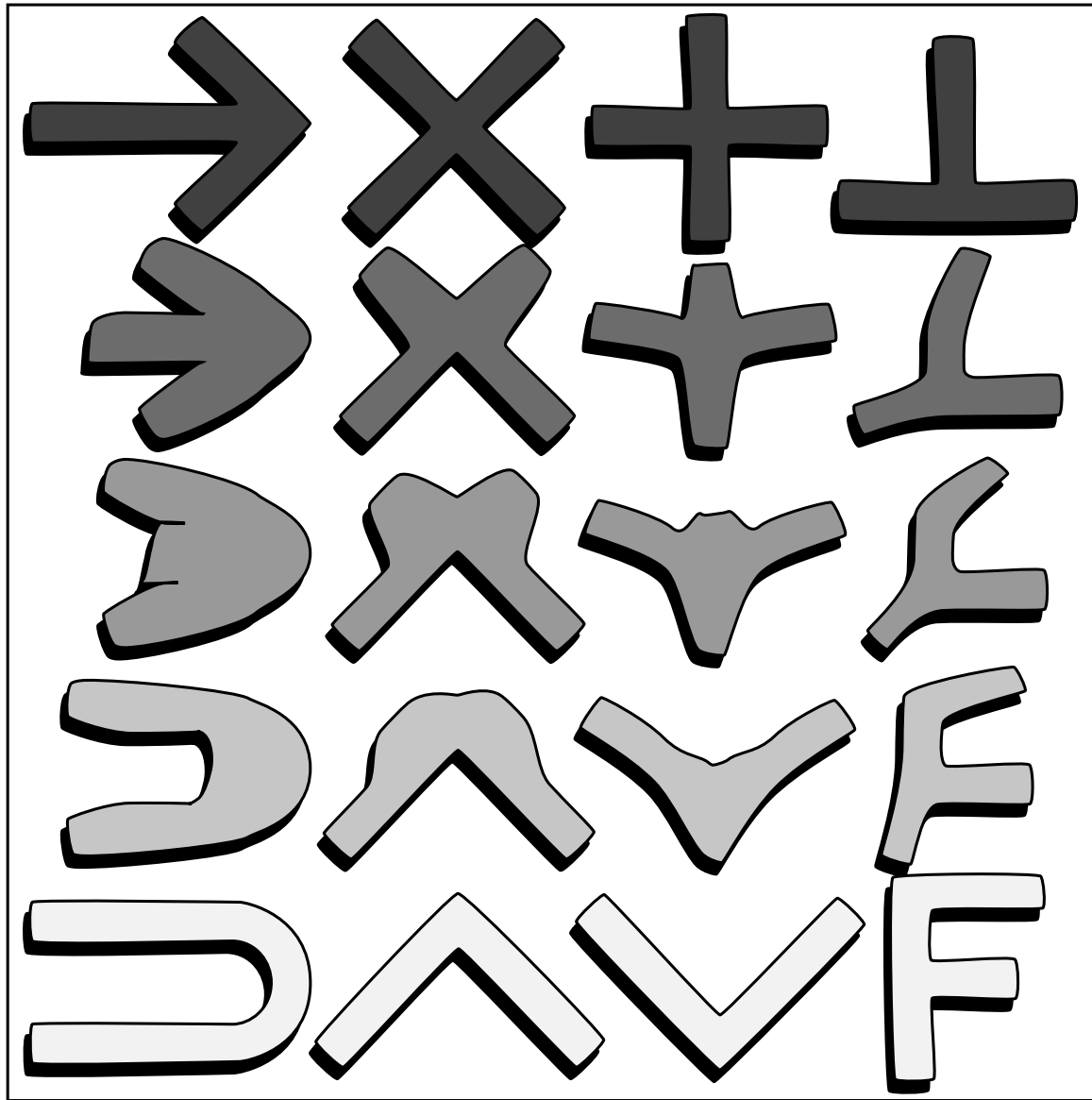
Subformula principle

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 Γ is a subformula of A or of some formula of Γ .

— Dag Prawitz, 1965



LC'90

The Curry-Howard homeomorphism

Alonzo Church (1903–1995)



Typed λ -calculus

$$\frac{\begin{array}{c} [x : A]^x \\ \vdots \\ N : B \end{array}}{\lambda x. N : A \rightarrow B} \rightarrow\text{-I}^x \qquad \frac{L : A \rightarrow B \quad M : A}{LM : B} \rightarrow\text{-E}$$

$$\frac{M : A \quad N : B}{(M, N) : A \times B} \times\text{-I} \qquad \frac{L : A \times B}{\mathbf{fst} L : A} \times\text{-E}_0 \qquad \frac{L : A \times B}{\mathbf{snd} L : B} \times\text{-E}_1$$

Normalising terms

$$\frac{\frac{[x : A]^x \quad \vdots \quad N : B}{\lambda x. N : A \rightarrow B} \rightarrow\text{-I}^x \quad \frac{\vdots \quad M : A}{\quad} \rightarrow\text{-E}}{(\lambda x. N) M : B} \rightarrow\text{-E} \quad \Longrightarrow \quad \frac{\vdots \quad M : A}{\quad} N\{M/x\} : B$$

$$\frac{\frac{\frac{\vdots \quad M : A \quad \vdots \quad N : B}{(M, N) : A \times B} \times\text{-I}}{\text{fst}(M, N) : A} \times\text{-E}_0 \quad \Longrightarrow \quad \frac{\vdots}{M : A}$$

Normalisation

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

$$\{\overline{\ell = M}\}.\ell_i \rightsquigarrow M_i$$

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

$$\mathbf{for } y \mathbf{ in } (\mathbf{for } x \mathbf{ in } L \mathbf{ do } M) \mathbf{ do } N \rightsquigarrow \mathbf{for } x \mathbf{ in } L \mathbf{ do } (\mathbf{for } y \mathbf{ in } M \mathbf{ do } N)$$

$$\mathbf{for } x \mathbf{ in } (\mathbf{if } L \mathbf{ then } M) \mathbf{ do } N \rightsquigarrow \mathbf{if } L \mathbf{ then } (\mathbf{for } x \mathbf{ in } M \mathbf{ do } N)$$

$$\mathbf{for } x \mathbf{ in } [] \mathbf{ do } N \rightsquigarrow []$$

$$\mathbf{for } x \mathbf{ in } (L @ M) \mathbf{ do } N \rightsquigarrow (\mathbf{for } x \mathbf{ in } L \mathbf{ do } N) @ (\mathbf{for } x \mathbf{ in } M \mathbf{ do } N)$$

$$\mathbf{if } \mathbf{true} \mathbf{ then } M \rightsquigarrow M$$

$$\mathbf{if } \mathbf{false} \mathbf{ then } M \rightsquigarrow []$$

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

Nested intermediate data

Flat data

departments

dpt
"Product"
"Quality"
"Research"
"Sales"

employees

dpt	emp
"Product"	"Alex"
"Product"	"Bert"
"Research"	"Cora"
"Research"	"Drew"
"Research"	"Edna"
"Sales"	"Fred"

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"design"
"Drew"	"abstract"
"Drew"	"design"
"Edna"	"abstract"
"Edna"	"call"
"Edna"	"design"
"Fred"	"call"

Importing the database

```
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

```
[ {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

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

```
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 XPath to SQL

Part VII

Results

SQL LINQ results (F#)

Example	F# 2.0	F# 3.0	us	(norm)
differences	17.6	20.6	18.1	0.5
range	×	5.6	2.9	0.3
satisfies	2.6	×	2.9	0.3
P(t ₀)	2.8	×	3.3	0.3
P(t ₁)	2.7	×	3.0	0.3
expertise'	7.2	9.2	8.0	0.6
expertise	×	66.7 ^{av}	8.3	0.9
xp ₀	×	8.3	7.9	1.9
xp ₁	×	14.7	13.4	1.1
xp ₂	×	17.9	20.7	2.2
xp ₃	×	3744.9	3768.6	4.4

Times in milliseconds; ^{av} marks query avalanche.

Feldspar results (Haskell)

	QDSL Feldspar		EDSL Feldspar		Generated Code	
	Compile	Run	Compile	Run	Compile	Run
IPGray	16.96	0.01	15.06	0.01	0.06	0.39
IPBW	17.08	0.01	14.86	0.01	0.06	0.19
FFT	17.87	0.39	15.79	0.09	0.07	3.02
CRC	17.14	0.01	15.33	0.01	0.05	0.12
Window	17.85	0.02	15.77	0.01	0.06	0.27

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.

A practical theory of language integrated query,
Cheney, Lindley, Wadler, ICFP, 2013.

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

Propositions as types, Wadler, CACM, Dec 2015.

<http://fsprojects.github.io/FSharp.Linq.Experimental.ComposableQuery/>



Ezra Cooper^{*†}, James Cheney^{*}, Sam Lindley^{*},
Shayan Najd^{*‡}, Josef Svenningsson[§], Philip Wadler^{*}

^{*}University of Edinburgh, [†]Qumulo, [‡]Google, [§]Chalmers & HiQ