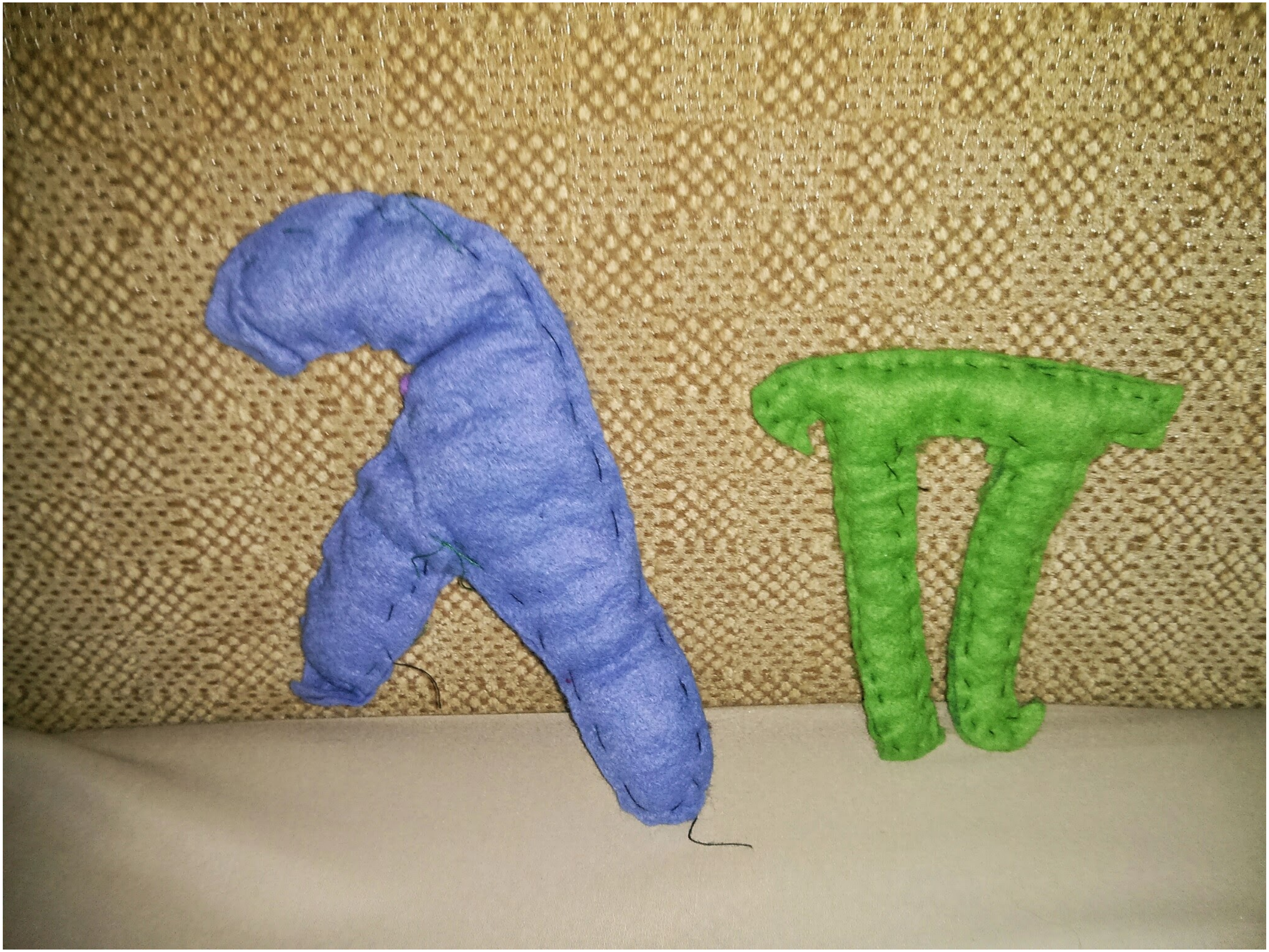
Everything old is new again: Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

DSLDISS
Lausanne, Monday 13 July 2015


Mayer
THERE'S
SCIESZKA / SMITH
WHEN SHEEP
CRONIN



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

$$
\begin{array}{l}
\{\mathsf{people} = \\
\quad [\,\{\mathsf{name} = \text{“Alex”}\,;\ \mathsf{age} = 40\};\\
\quad\ \ \{\mathsf{name} = \text{“Bert”}\,;\ \mathsf{age} = 30\};\\
\quad\ \ \{\mathsf{name} = \text{“Cora”}\,;\ \mathsf{age} = 35\};\\
\quad\ \ \{\mathsf{name} = \text{“Drew”};\ \mathsf{age} = 60\};\\
\quad\ \ \{\mathsf{name} = \text{“Edna”};\ \mathsf{age} = 25\};\\
\quad\ \ \{\mathsf{name} = \text{“Fred”}\,;\ \mathsf{age} = 70\}\,]\,\}
\end{array}
$$

A query as F# code (naive)

$\textbf{type}\ \mathsf{DB} = \{\mathsf{people} : \{\mathsf{name} : \textbf{string};\ \mathsf{age} : \textbf{int}\}\ \textbf{list}\}$

$\textbf{let}\ \mathsf{db}' : \mathsf{DB} = \textbf{database}(\text{“People”})$

$\textbf{let}\ \mathsf{youths}' : \{\mathsf{name} : \textbf{string};\ \mathsf{age} : \textbf{int}\}\ \textbf{list} =$

$\quad \textbf{for}\ \mathsf{u}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do}$

$\quad \textbf{for}\ \mathsf{v}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do}$

$\quad \textbf{if}\ \mathsf{u.name} = \text{“Alex”}\ \&\&\ \mathsf{v.age} < \mathsf{u.age}\ \textbf{then}$

$\quad \textbf{yield}\ \{\mathsf{name} : \mathsf{v.name}; \mathsf{age} : \mathsf{v.age}\}$

$\mathsf{youths}' \rightsquigarrow$

$\quad [\{\mathsf{name} = \text{“Bert”} ;\ \mathsf{age} = 30\}$

$\quad \{\mathsf{name} = \text{“Cora”} ;\ \mathsf{age} = 35\}$

$\quad \{\mathsf{name} = \text{“Edna”};\ \mathsf{age} = 25\}]$

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) ⇝
  [{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)

$$\begin{array}{l}
\textbf{val}\ \mathsf{youth}' : \mathsf{List}\,[\,\{\,\textbf{val}\ \mathsf{name} : \mathsf{String};\ \textbf{val}\ \mathsf{age} : \mathsf{Int}\}\,]\ = \\
\quad \textbf{for}\ \{\mathsf{u} \leftarrow \mathsf{db}'.\mathsf{people} \\
\qquad\quad \mathsf{v} \leftarrow \mathsf{db}'.\mathsf{people} \\
\qquad\quad \textbf{if}\ \mathsf{u.name}\ ==\ \text{“Alex”}\ \&\&\ \mathsf{v.age} < \mathsf{u.age}\} \\
\quad \textbf{yield new}\ \mathsf{Record}\ \{\ \textbf{val}\ \mathsf{name} = \mathsf{v.name};\ \textbf{val}\ \mathsf{age} = \mathsf{v.age}\ \}
\end{array}$$

Scala (quoted)

$$\begin{array}{l}
\textbf{val}\ \mathsf{youth} : \mathsf{Rep}\,[\ \mathsf{List}\,[\,\{\,\textbf{val}\ \mathsf{name} : \mathsf{String};\ \textbf{val}\ \mathsf{age} : \mathsf{Int}\}\,]\,]\ = \\
\quad \textbf{for}\ \{\mathsf{u} \leftarrow \mathsf{db}.\mathsf{people} \\
\qquad\quad \mathsf{v} \leftarrow \mathsf{db}.\mathsf{people} \\
\qquad\quad \textbf{if}\ \mathsf{u.name}\ ==\ \text{“Alex”}\ \&\&\ \mathsf{v.age} < \mathsf{u.age}\} \\
\quad \textbf{yield new}\ \mathsf{Record}\ \{\ \textbf{val}\ \mathsf{name} = \mathsf{v.name};\ \textbf{val}\ \mathsf{age} = \mathsf{v.age}\ \}
\end{array}$$

Part II

Abstraction, composition, dynamic generation

Abstracting over values

$$\textbf{let}\ \text{range} : \text{Expr}{<}\,(\textbf{int}, \textbf{int}) \to \text{Names}\,{>} =$$

$$\texttt{<@}\ \textbf{fun}(a, b) \to \textbf{for}\ w\ \textbf{in}\ (\%db).\text{people}\ \textbf{do}$$

$$\textbf{if}\ a \le w.\text{age}\ \ \&\&\ \ w.\text{age} < b\ \textbf{then}$$

$$\textbf{yield}\ \{\text{name} : w.\text{name}\}\ \texttt{@>}$$

$$\textbf{run}(\texttt{<@}\ (\%\text{range})(30, 40)\ \texttt{@>})$$

$$\textbf{select}\ w.\text{name}\ \textbf{as}\ \text{name}$$

$$\textbf{from}\ \text{people}\ \textbf{as}\ w$$

$$\textbf{where}\ 30 \le w.\text{age}\ \textbf{and}\ w.\text{age} < 40$$

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

$$\begin{array}{l}\textbf{type}\ \text{Predicate} = \\ \quad \mid\ \text{Above}\ \textbf{of int} \\ \quad \mid\ \text{Below}\ \textbf{of int} \\ \quad \mid\ \text{And}\ \textbf{of}\ \text{Predicate} \times \text{Predicate}\end{array}$$

$$\begin{array}{l}\textbf{let rec}\ \text{P}(\text{t} : \text{Predicate}) : \text{Expr}{<}\,\textbf{int} \rightarrow \textbf{bool}\,{>} = \\ \quad \textbf{match}\ \text{t}\ \textbf{with} \\ \quad \mid\ \text{Above(a)} \rightarrow \texttt{<@}\ \textbf{fun}(\text{x}) \rightarrow (\%\textbf{lift}(\text{a})) \leq \text{x}\ \texttt{@>} \\ \quad \mid\ \text{Below(a)} \rightarrow \texttt{<@}\ \textbf{fun}(\text{x}) \rightarrow \text{x} < (\%\textbf{lift}(\text{a}))\ \texttt{@>} \\ \quad \mid\ \text{And(t, u)} \rightarrow \texttt{<@}\ \textbf{fun}(\text{x}) \rightarrow (\%\text{P(t)})(\text{x})\ \ \&\&\ \ (\%\text{P(u)})(\text{x})\ \texttt{@>}\end{array}$$

Dynamically generated queries

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

$$\rightsquigarrow \texttt{<@}\ \ \textbf{fun}(x) \rightarrow (\textbf{fun}(x_1) \rightarrow 30 \leq x_1)(x)\ \ \&\&\ \ (\textbf{fun}(x_2) \rightarrow x_2 < 40)(x)\ \ \texttt{@>}$$

$$\rightsquigarrow \texttt{<@}\ \ \textbf{fun}(x) \rightarrow 30 \leq x\ \ \&\&\ \ x < 40\ \ \texttt{@>}$$

$$\textbf{run}(\texttt{<@}\ (\%\mathrm{satisfies})(\%\mathrm{P}(\mathrm{And}(\mathrm{Above}(30), \mathrm{Below}(40))))\ \texttt{@>})$$

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

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

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

vs.

$\textbf{let}\ \mathsf{satisfies'}(\mathsf{p} : \mathsf{Expr}\!<\textbf{int}> \to \mathsf{Expr}\!<\textbf{bool}>) : \mathsf{Expr}\!<\mathsf{Names}> =$
$\quad$ `<@` $\textbf{for}\ \mathsf{w}\ \textbf{in}\ (\%\mathsf{db}).\mathsf{people}\ \textbf{do}$
$\quad\quad \textbf{if}\ (\%\mathsf{p}($`<@` $\mathsf{w.age}$ `@>`$))\ \textbf{then}$
$\quad\quad \textbf{yield}\ \{\mathsf{name} : \mathsf{w.name}\}$ `@>`

QDSL		EDSL	
$\mathsf{Expr}< \mathsf{A} \rightarrow \mathsf{B} >$	✔	$\mathsf{Expr}< \mathsf{A} > \rightarrow \mathsf{Expr}< \mathsf{B} >$	✔
$\mathsf{Expr}< \mathsf{A} \times \mathsf{B} >$	✔	$\mathsf{Expr}< \mathsf{A} > \times \mathsf{Expr}< \mathsf{B} >$	✔
$\mathsf{Expr}< \mathsf{A} + \mathsf{B} >$	✔	$\mathsf{Expr}< \mathsf{A} > + \mathsf{Expr}< \mathsf{B} >$	✗

closed quotations

vs.

open quotations

quotations of functions

$(\mathsf{Expr}\!<\mathsf{A} \longrightarrow \mathsf{B}>)$

vs.

functions of quotations

$(\mathsf{Expr}\!<\mathsf{A}> \longrightarrow \mathsf{Expr}\!<\mathsf{B}>)$

Part IV

The Subformula Principle

Gerhard Gentzen (1909–1945)

Natural Deduction — Gentzen (1935)

&–I

$$\frac{\mathfrak{A} \qquad \mathfrak{B}}{\mathfrak{A} \,\&\, \mathfrak{B}}$$

&–E

$$\frac{\mathfrak{A} \,\&\, \mathfrak{B}}{\mathfrak{A}} \qquad \frac{\mathfrak{A} \,\&\, \mathfrak{B}}{\mathfrak{B}}$$

∨–I

$$\frac{\mathfrak{A}}{\mathfrak{A} \vee \mathfrak{B}} \qquad \frac{\mathfrak{B}}{\mathfrak{A} \vee \mathfrak{B}}$$

∨–E

$$\frac{\mathfrak{A} \vee \mathfrak{B} \qquad \begin{matrix}[\mathfrak{A}] \\ \mathfrak{C}\end{matrix} \qquad \begin{matrix}[\mathfrak{B}] \\ \mathfrak{C}\end{matrix}}{\mathfrak{C}}$$

∀–I

$$\frac{\mathfrak{F}a}{\forall \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}$$

∀–E

$$\frac{\forall \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}{\mathfrak{F}a}$$

∃–I

$$\frac{\mathfrak{F}a}{\exists \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}$$

∃–E

$$\frac{\exists \mathfrak{x}\, \mathfrak{F}\mathfrak{x} \qquad \begin{matrix}[\mathfrak{F}a] \\ \mathfrak{C}\end{matrix}}{\mathfrak{C}}$$

⊃–I

$$\frac{\begin{matrix}[\mathfrak{A}] \\ \mathfrak{B}\end{matrix}}{\mathfrak{A} \supset \mathfrak{B}}$$

⊃–E

$$\frac{\mathfrak{A} \qquad \mathfrak{A} \supset \mathfrak{B}}{\mathfrak{B}}$$

¬–I

$$\frac{\begin{matrix}[\mathfrak{A}] \\ \curlywedge\end{matrix}}{\neg \mathfrak{A}}$$

¬–E

$$\frac{\mathfrak{A} \qquad \neg \mathfrak{A}}{\curlywedge} \qquad \frac{\curlywedge}{\mathfrak{D}} \, .$$

Natural Deduction

$$\dfrac{\begin{matrix}[A]^x \\ \vdots \\ B\end{matrix}}{A \supset B} \supset\text{-I}^x \qquad \dfrac{A \supset B \qquad A}{B} \supset\text{-E}$$

$$\dfrac{A \qquad B}{A \,\&\, B} \,\&\text{-I} \qquad \dfrac{A \,\&\, B}{A} \,\&\text{-E}_0 \qquad \dfrac{A \,\&\, B}{B} \,\&\text{-E}_1$$

Proof Normalisation

$$\dfrac{\dfrac{\begin{matrix}[A]^x\\ \vdots \\ B\end{matrix}}{A \supset B}\,\supset\text{-I}^x \qquad \begin{matrix}\vdots\\ A\end{matrix}}{B}\,\supset\text{-E} \qquad\Longrightarrow\qquad \begin{matrix}\vdots\\ A\\ \vdots\\ B\end{matrix}$$

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ A\end{matrix} \qquad \begin{matrix}\vdots\\ B\end{matrix}}{A \,\&\, B}\,\&\text{-I}}{A}\,\&\text{-E}_0 \qquad\Longrightarrow\qquad \begin{matrix}\vdots\\ A\end{matrix}$$

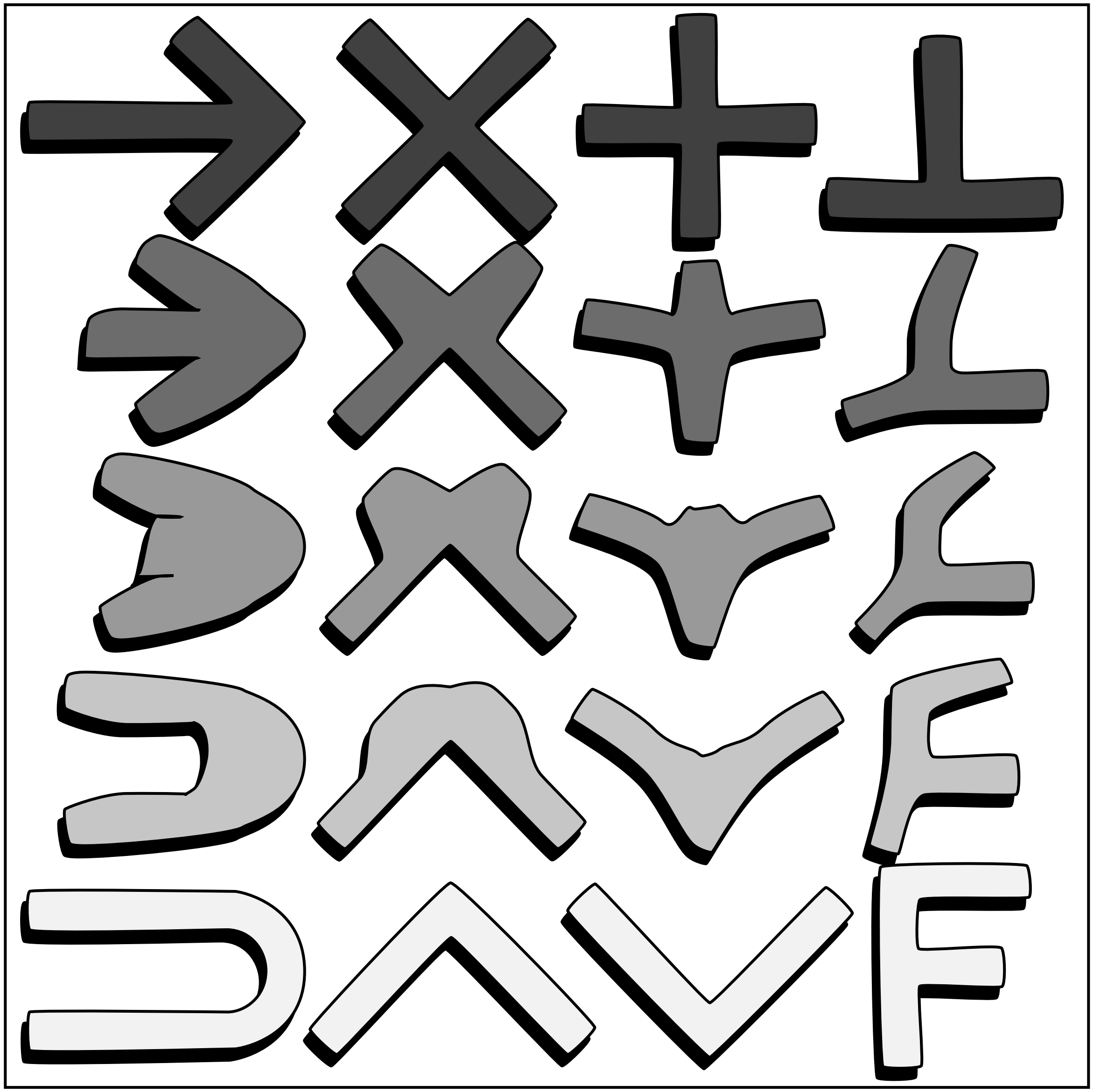
Subformula principle

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 occuring 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

$$\dfrac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x.\, N : A \to B}\ {\to}\text{-I}^x \qquad \dfrac{L : A \to B \qquad M : A}{L\, M : B}\ {\to}\text{-E}$$

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

Normalising terms

$$\dfrac{\dfrac{\begin{array}{c}[x : A]^x\\ \vdots\\ N : B\end{array}}{\lambda x.\,N : A \to B}\,{\to}\text{-I}^x \qquad \begin{array}{c}\vdots\\ M : A\end{array}}{(\lambda x.\,N)\,M : B}\,{\to}\text{-E} \qquad\Longrightarrow\qquad \begin{array}{c}\vdots\\ M : A\\ \vdots\\ N\{M/x\} : B\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ M : A\end{array} \qquad \begin{array}{c}\vdots\\ N : B\end{array}}{(M, N) : A \times B}\,{\times}\text{-I}}{\textbf{fst}\,(M, N) : A}\,{\times}\text{-E}_0 \qquad\Longrightarrow\qquad \begin{array}{c}\vdots\\ M : A\end{array}$$

Normalisation

$$
\begin{array}{rcl}
(\textbf{fun}(x) \rightarrow N)\ M & \rightsquigarrow & N[x := M] \\
\{\overline{\ell = M}\}.\ell_i & \rightsquigarrow & M_i \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{yield}\ M)\ \textbf{do}\ N & \rightsquigarrow & N[x := M] \\
\textbf{for}\ y\ \textbf{in}\ (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M)\ \textbf{do}\ N & \rightsquigarrow & \textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (\textbf{for}\ y\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{if}\ L\ \textbf{then}\ M)\ \textbf{do}\ N & \rightsquigarrow & \textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ [\ ]\ \textbf{do}\ N & \rightsquigarrow & [\ ] \\
\textbf{for}\ x\ \textbf{in}\ (L\ \texttt{@}\ M)\ \textbf{do}\ N & \rightsquigarrow & (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N)\ \texttt{@}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{if true then}\ M & \rightsquigarrow & M \\
\textbf{if false then}\ M & \rightsquigarrow & [\ ]
\end{array}
$$

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

$$\begin{aligned}
&\textbf{type}\ \mathsf{Org} = \{\mathsf{departments} : \{\mathsf{dpt} : \textbf{string}\}\ \textbf{list};\\
&\qquad\qquad\quad \mathsf{employees} : \{\mathsf{dpt} : \textbf{string}; \mathsf{emp} : \textbf{string}\}\ \textbf{list};\\
&\qquad\qquad\quad \mathsf{tasks} : \{\mathsf{emp} : \textbf{string}; \mathsf{tsk} : \textbf{string}\}\ \textbf{list}\ \}\\
&\textbf{let}\ \mathsf{org} : \mathsf{Expr}\!<\mathsf{Org}> = \texttt{<@}\ \textbf{database}(\text{“Org”})\ \texttt{@>}
\end{aligned}$$

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

$$
\begin{array}{l}
[\{\mathsf{dpt} = \text{“Product”};\ \mathsf{employees} = \\
\quad [\{\mathsf{emp} = \text{“Alex”};\ \mathsf{tasks} = [\text{“build”}]\} \\
\quad \{\mathsf{emp} = \text{“Bert”};\ \mathsf{tasks} = [\text{“build”}]\}]\}; \\
\{\mathsf{dpt} = \text{“Quality”}; \mathsf{employees} = [\,]\}; \\
\{\mathsf{dpt} = \text{“Research”};\ \mathsf{employees} = \\
\quad [\{\mathsf{emp} = \text{“Cora”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“build”}; \text{“design”}]\}; \\
\quad \{\mathsf{emp} = \text{“Drew”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“design”}]\}; \\
\quad \{\mathsf{emp} = \text{“Edna”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“call”}; \text{“design”}]\}]\}; \\
\{\mathsf{dpt} = \text{“Sales”};\ \mathsf{employees} = \\
\quad [\{\mathsf{emp} = \text{“Fred”};\ \mathsf{tasks} = [\text{“call”}]\}]\}]
\end{array}
$$

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

$$\textbf{let}\ \text{expertise} : \text{Expr}{<}\ \textbf{string} \rightarrow \{\text{dpt} : \textbf{string}\}\ \textbf{list}\ {>} =$$

```
<@ 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	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,
Cheney, Lindley, Wadler, ICFP, 2013.

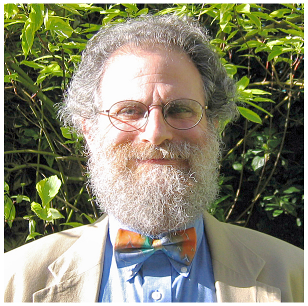
Everything old is new again: Quoted Domain Specific Languages,
Najd, Lindley, Svenningsson, Wadler, Draft, 2015.

Propositions as types, Wadler, CACM, to appear.

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





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