First-Class Functions For First-Order Database Engines

Torsten Grust Alexander Ulrich

Database Systems @ Universität Tübingen

DBPL 2013, August 30, 2013

Widely available and implemented database query and programming languages are first-order today.

- User-defined functions merely are units of query and program organization
- User-defined functions are not considered first-class values

Wait! There is XQuery 3.0

- XQuery 3.0 takes a major step towards a full-fledged functional language

```
3 Expressions
   3.1 Primary Expressions
      3.1.1 Literals
      ...
      3.1.6 Named Function References
      3.1.7 Inline Function Expressions
   3.2 Postfix Expressions
      3.2.1 Filter Expressions
      3.2.2 Dynamic Function Call
```

- But major/commercial database-based implementations do not follow

Wait! There is XQuery 3.0

- XQuery 3.0 takes a major step towards a full-fledged functional language

```
3 Expressions
   3.1 Primary Expressions
      3.1.1 Literals
      ...
      3.1.6 Named Function References
      3.1.7 Inline Function Expressions
   3.2 Postfix Expressions
      3.2.1 Filter Expressions
      3.2.2 Dynamic Function Call
```

- But major/commercial database-based implementations ~~do not~~ cannot follow

And Then There is PL/SQL...

- “Hopelessly first-order”; tied to value types of the underlying database engine
- “Stored procedures” are mere code units, defined and named at compile time
- only static function calls admissable

If I Had First-Class Functions...

- The function type

  ```
  FUNCTION(t1) RETURNS t2
  ```

  would be a data type much like `INTEGER`, `VARCHAR(t)`

- I would create tables holding function-typed columns:

```
CREATE TABLE funs (id INTEGER,
                   fn FUNCTION(real) RETURNS real)
```

If I Had First-Class Functions...

- I would insert (references to) built-in and user-defined functions as well as literal functions into tables:

```
INSERT INTO funs VALUES
  (1, atan),
  (2, square),
  (3, FUNCTION (x real) RETURNS real AS
        BEGIN RETURN 2 * x; END);
```

If I Had First-Class Functions...

- I would naturally map functional concepts onto functions, inside my queries:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad h \to 0$$

If I Had First-Class Functions...

- I would naturally map functional concepts onto functions, inside my queries:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad h \to 0$$

```
-- real -> (real -> real) -> (real -> real)
CREATE FUNCTION diffq(h real, f FUNCTION(real) RETURNS real)
    RETURNS FUNCTION(real) RETURNS real AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;
```

If I Had First-Class Functions...

- I would use queries to dynamically route functions and arguments

```
SELECT id, x,
       fn(x) AS fx,
       derive(fn)(x) AS "f'x"
FROM   FUNS, ARGS;
```

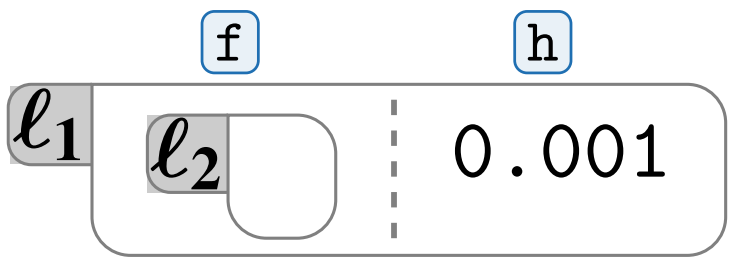
Most database engines are not only first-order but also closed black boxes.

Aim for a non-invasive approach that turns first-class functions into first-order regular values.

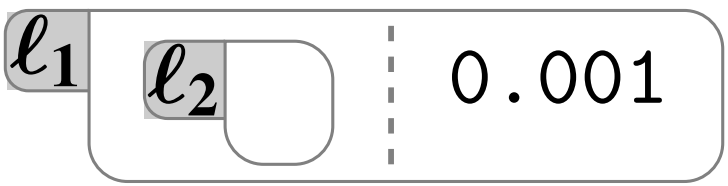
Query Defunctionalization

Closures

```
CREATE FUNCTION diffq(h real, f FUNCTION(real) RETURNS real)
    RETURNS FUNCTION(real) RETURNS real AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;
```



Representing Closures (1)



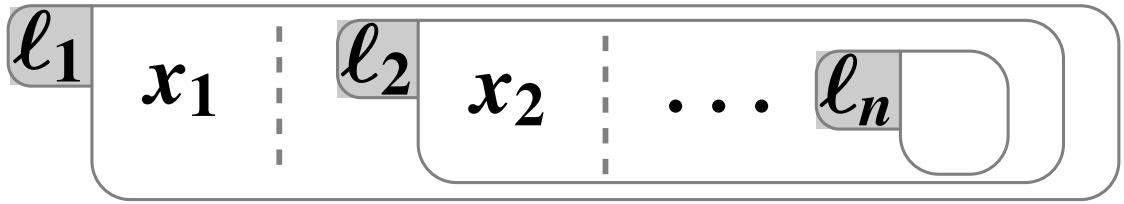
$\Rightarrow$

```
CREATE TYPE clos1 AS (
    label label,
    env1 clos2,
    env2 real);
```

+

ROW(ℓ_1, ROW(ℓ_2), 0.001)

Representing Closures (2)



```
CREATE TYPE (
  label label,
  key int);
```

+ ROW(ℓ_1, γ_1) +

envtab

id	env
γ_1	ROW(x_1, ROW(ℓ_2, γ_2))
γ_2	ROW(x_2, ROW(ℓ_3, γ_3))
⋮	⋮
γ_n	ROW(x_n, NULL)

Where Does the Function Body Go?

```
CREATE FUNCTION diffq(...) RETURNS ... AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;
```

Where Does the Function Body Go?

```
CREATE FUNCTION diffq(...) RETURNS ... AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;
```

Where Does the Function Body Go?

```
CREATE FUNCTION diffq(...) RETURNS ... AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;
```


```
CREATE FUNCTION ℓ1(x real, f clos, h real) RETURNS real AS
BEGIN
  RETURN (dispatch(f, x + h) - dispatch(f, x)) / h;
END;
```

Relate Closures and Code

```
CREATE FUNCTION dispatch(clos clos, g real) RETURNS real AS
BEGIN
  -- get environment from table using clos.key -> env
  CASE clos.label
    WHEN 'ℓ1' RETURN ℓ1(g, env1, env2);
    WHEN 'ℓ2' RETURN ...
END;
```

Relate Closures and Code

```
CREATE FUNCTION dispatch(clos clos, g real) RETURNS real AS
BEGIN
  -- get environment from table using clos.key -> env
  CASE clos.label
    WHEN 'ℓ1' RETURN ℓ1(g, env1, env2);
    WHEN 'ℓ2' RETURN ...
END;
```

```
      f(x + h)
```

```
f ::  real → real
```

Relate Closures and Code

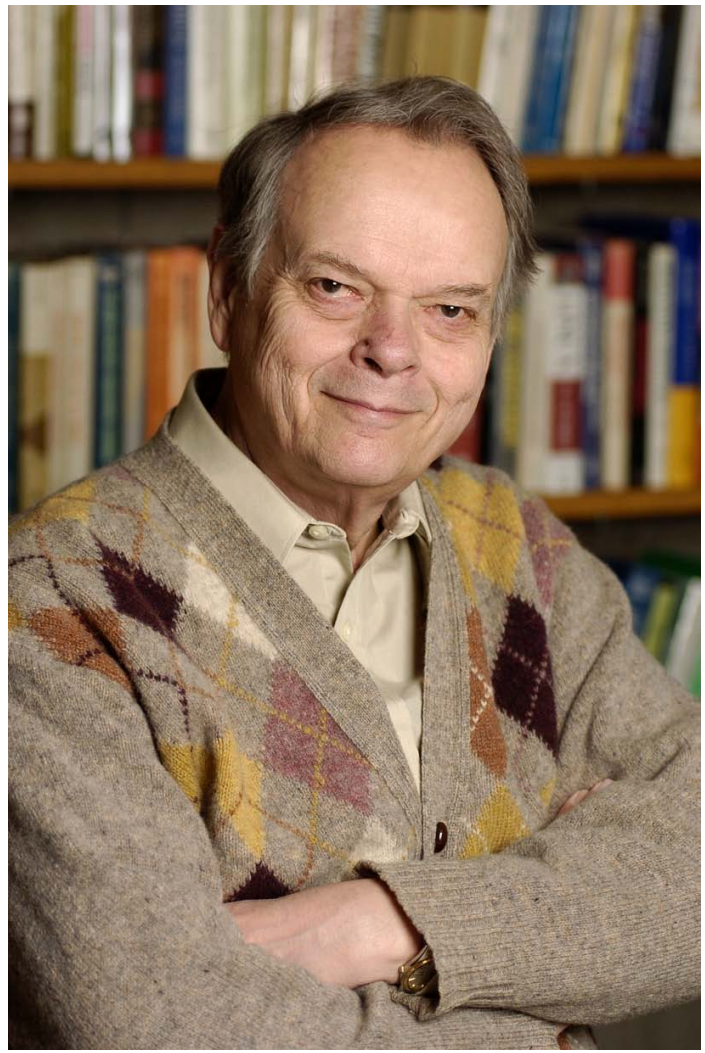
```
CREATE FUNCTION dispatch(clos clos, g real) RETURNS real AS
BEGIN
  -- get environment from table using clos.key -> env
  CASE clos.label
    WHEN 'ℓ1' RETURN ℓ1(g, env1, env2);
    WHEN 'ℓ2' RETURN ...
END;
```

$$\texttt{f(x + h)} \quad \Longrightarrow \quad \texttt{dispatch(f, x + h)}$$

$$\texttt{f :: real} \to \texttt{real} \qquad\qquad \texttt{f :: clos}$$

PL/SQL: Typed Closures

Defunctionalization



John C. Reynolds (1935 – 2013)

Query Defunctionalization

First-Class Functions	First-Order Database Engine
Literal Function	Closure Constructor

Query Defunctionalization

First-Class Functions	First-Order Database Engine
Literal Function	Closure Constructor
Named Function Reference	Empty Closure Constructor

Query Defunctionalization

First-Class Functions	**First-Order Database Engine**
Literal Function	Closure Constructor
Named Function Reference	Empty Closure Constructor
Dynamic Function Call	Static `dispatch()` Call

Query Defunctionalization: Syntactic Whole-Query Transformation

```
-- approximate derivative: the differential quotient for function f
CREATE FUNCTION diffq(h real, f FUNCTION(real) RETURNS real)
    RETURNS FUNCTION(real) RETURNS real AS
BEGIN
  RETURN FUNCTION(x real) RETURNS real AS
         BEGIN
             RETURN (f(x + h) - f(x)) / h;
         END;
END;

-- compute first derivative of function f
CREATE FUNCTION derive(f FUNCTION(real) RETURNS real)
    RETURNS FUNCTION(real) RETURNS real AS
BEGIN
    RETURN diffq(0.001, f); -- fix a small h, here: 0.001
END;

CREATE FUNCTION square(x real) RETURNS real AS
BEGIN
  RETURN x * x;
END;

CREATE TABLE IF NOT EXISTS funs (id integer NOT NULL PRIMARY KEY,
                                 fn FUNCTION(real) RETURNS real);
INSERT INTO funs VALUES
(1, atan),    -- built-in function
(2, square), -- user-def function
(3, FUNCTION(x real) RETURNS real AS BEGIN RETURN 2 * x; END);

CREATE TABLE args (x real NOT NULL);
INSERT INTO args VALUES (-100.0),(-99.0), ...,
    (-1.0), (0.0), (1.0), (2.0), ..., (99.0),(100.0);

-- tabulation of all functions along with their first derivatives
SELECT id, x, fn(x) AS fx, derive(fn)(x) AS "f'x"
FROM   funs, args;
```

```
CREATE TYPE label1 AS ENUM ('fun1_4', 'fun1_3', 'fun1_2', 'fun1_1');
CREATE TYPE clos1 AS (label label1, closkey int); -- real → real closures
CREATE TABLE
envtab1 (closkey serial PRIMARY KEY, env1 clos1, env2 real);

CREATE OR REPLACE FUNCTION closconst1(label label1, env1 clos1, env2 real) RETURNS clos1 AS
DECLARE
    key int ;
BEGIN
    INSERT INTO envtab1 (env1, env2) VALUES (env1, env2) RETURNING closkey INTO key;
    RETURN ROW(label, key);
END;

CREATE FUNCTION dispatch1(clos clos1, b1 real) RETURNS real AS
DECLARE
    env envtab1 ;
BEGIN
    SELECT * INTO env FROM envtab1 et WHERE et.closkey = clos.closkey;
    CASE clos.label
        WHEN 'fun1_4' THEN RETURN lifted2(b1, env.env1, env.env2);
        WHEN 'fun1_3' THEN RETURN lifted1(b1);
        WHEN 'fun1_2' THEN RETURN square(b1);
        WHEN 'fun1_1' THEN RETURN atan(b1);
    END CASE;
END;

CREATE FUNCTION lifted2(x real, f clos1, h real) RETURNS real AS
BEGIN
    RETURN (dispatch1(f, x + h) - dispatch1(f, x)) / h;
END;

CREATE FUNCTION lifted1(x real) RETURNS real AS
BEGIN
    RETURN 2 * x;
END;

[ CREATE FUNCTION square(x real) RETURNS real AS ... ]

CREATE TABLE IF NOT EXISTS funs (id int NOT NULL PRIMARY KEY, fn clos1);
INSERT INTO funs VALUES
(1, ROW('fun1_1', NULL)),
-- atan
(2, ROW('fun1_2', NULL)),
-- square
(3, ROW('fun1_3', NULL));
-- function literal '2 * x'

CREATE FUNCTION diffq(h real, f clos1)
    RETURNS clos1 AS
BEGIN
    RETURN closconst1('fun1_4', f, h);
END;

CREATE FUNCTION derive(f clos1) RETURNS clos1 AS
BEGIN
    RETURN diffq(1.0e-3, f);
END;

CREATE TABLE args (x real NOT NULL);
INSERT INTO args VALUES (-100.0), (-99.0), ..., (99.0), (100.0);

SELECT id, x, dispatch1(fn, x) as fx, dispatch1(derive(fn), x) AS "f'x"
FROM funs, args;
```

Query Defunctionalization: Introduces Tolerable Runtime Overhead

- Inlining `dispatch()`
- Avoid `dispatch()` at all
- Construct closures wisely
- Avoid closure construction

PL/SQL + XQuery + 

Details In The Paper

$$\mathcal{D}[\![\texttt{declare function}\ name(\$x_1,\ldots,\$x_n)\ \{\ e\ \}]\!] = \texttt{declare function}\ name(\$x_1,\ldots,\$x_n)\ \{\ \mathcal{E}[\![e]\!]\ \}$$

$$\begin{aligned}
\mathcal{E}[\![\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2]\!] &= \texttt{for}\ \$v\ \texttt{in}\ \mathcal{E}[\![e_1]\!]\ \texttt{return}\ \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![\texttt{let}\ \$v := e_1\ \texttt{return}\ e_2]\!] &= \texttt{let}\ \$v := \mathcal{E}[\![e_1]\!]\ \texttt{return}\ \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![\$v]\!] &= \$v \\
\mathcal{E}[\![\texttt{if}\ (e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &= \texttt{if}\ (\mathcal{E}[\![e_1]\!])\ \texttt{then}\ \mathcal{E}[\![e_2]\!]\ \texttt{else}\ \mathcal{E}[\![e_3]\!] \\
\mathcal{E}[\![(e_1,\ldots,e_n)]\!] &= (\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![e/a::t]\!] &= \mathcal{E}[\![e]\!]/a::t \\
\mathcal{E}[\![\texttt{element}\ n\ \{\ e\ \}]\!] &= \texttt{element}\ n\ \{\ \mathcal{E}[\![e]\!]\ \} \\
\mathcal{E}[\![e_1\,[e_2]]\!] &= \mathcal{E}[\![e_1]\!]\,[\mathcal{E}[\![e_2]\!]] \\
\mathcal{E}[\![.]\!] &= . \\
\mathcal{E}[\![name(e_1,\ldots,e_n)]\!] &= name(\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![\texttt{function}(\$x_1\ \texttt{as}\ t_1,\ldots,\$x_n\ \texttt{as}\ t_n)\ \texttt{as}\ t\ \{\ e\ \}]\!] &= \boxed{\ell\ \$v_1\ \cdots\ \$v_m} \qquad (1) \\
&\quad Dispatch(n) \leftarrow Dispatch(n) \cup \{branch\} \\
&\quad\quad Lifted \leftarrow Lifted \cup \{lifted\} \\
&\quad \textbf{where} \\
&\quad\quad \ell = label(n) \\
&\quad \$v_1,\ldots,\$v_m = fv(\texttt{function}(\$x_1\ \texttt{as}\ t_1,\ldots,\$x_n\ \texttt{as}\ t_n)\ \texttt{as}\ t\ \{\ e\ \}) \\
&\quad\quad branch = \boxed{\ell\ \$v_1\ \cdots\ \$v_m} \Rightarrow \\
&\quad\quad\quad\quad \ell(\$b_1,\ldots,\$b_n,\$v_1,\ldots,\$v_m) \\
&\quad\quad lifted = \texttt{declare function}\ \ell(\\
&\quad\quad\quad\quad \$x_1\ \texttt{as}\ t_1,\ldots,\$x_n\ \texttt{as}\ t_n,\$v_1,\ldots,\$v_m)\ \texttt{as}\ t \\
&\quad\quad\quad\quad \{\ \mathcal{E}[\![e]\!]\ \}; \\
\mathcal{E}[\![name\#n]\!] &= \boxed{\ell\ \ } \qquad (2) \\
&\quad Dispatch(n) \leftarrow Dispatch(n) \cup \{branch\} \\
&\quad \textbf{where} \\
&\quad\quad \ell = label(n) \\
&\quad branch = \boxed{\ell\ \ } \Rightarrow name(\$b_1,\ldots,\$b_n) \\
\mathcal{E}[\![e(e_1,\ldots,e_n)]\!] &= \texttt{dispatch_}n(\mathcal{E}[\![e]\!],\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]) \qquad (3)
\end{aligned}$$

$$\begin{aligned}
\mathcal{Q}[\![d_1;\ldots;d_n;e]\!] &= \forall\, i \in dom(Dispatch)\colon declare_dispatch(i, Dispatch(i)) \\
&\quad Lifted \\
&\quad \mathcal{D}[\![d_1]\!];\ldots;\mathcal{D}[\![d_n]\!];\,\mathcal{E}[\![e]\!]
\end{aligned}$$

In The Paper...

- Examples and use cases
  - functional maps
  - algebraic data types
  - flexible constraints
  - configurable queries
  - natural formulations (e.g. `group-by`)
- Transformation details for PL/SQL and XQuery
- Representation tweaks
- Optimizations
- Investigation of overhead

Query Defunctionalization

http://db.inf.uni-tuebingen.de