Functions Are Data Too

(Defunctionalization for PL/SQL)

Torsten Grust Nils Schweinsberg Alexander Ulrich

Universität Tübingen
Tübingen, Germany

[torsten.grust, alexander.ulrich]@uni-tuebingen.de
nils.schweinsberg@student.uni-tuebingen.de

ABSTRACT

We demonstrate a full-fledged implementation of *first-class functions* for the widely used PL/SQL database programming language. Functions are treated as regular data items that may be (1) constructed at query runtime, (2) stored in and retrieved from tables, (3) assigned to variables, and (4) passed to and from other (higher-order) functions. The resulting PL/SQL dialect concisely and elegantly expresses a wide range of new query idioms which would be cumbersome to formulate if functions remained second-class citizens. We include a diverse set of application scenarios that make these advantages tangible.

First-class PL/SQL functions require featherweight syntactic extensions only and come with a non-invasive implementation—the *defunctionalization* transformation—that can entirely be built on top of existing relational DBMS infrastructure. An interactive demonstrator helps users to experiment with the "*function as data*" paradigm and to earn a solid intuition of its inner workings.

1. FUNCTIONS ARE DATA TOO

PL/SQL programming [2] marks one of the predominant approaches to implement application logic close to relational data: regular SQL queries may be embedded in programs that feature—among other elements typically found in scripting languages—statement sequences, control flow and exception handling constructs, or variable assignment. Since the PL/SQL interpreter or compiler tightly integrates with the database engine, such programs can manipulate persistent data efficiently without crossing database kernel boundaries.

The colloquial term "*stored procedures*" is widely used as a stand-in for the PL/SQL approach as a whole and *functions* (or procedures) indeed are its primary unit of program organization. Yet, functions remain second-class citizens in the language: functions exclusively assume the role of code units, defined and named at compile time, ready for subsequent invocation.

Proceedings of the VLDB Endowment, Vol. 6, No. 12

In this demonstration (and a companion paper that zooms in on the conceptual details, implementation, and performance [4]), we explore a dialect of PL/SQL in which *functions assume the role of data* instead. As such, functions may be defined at query runtime, assigned to variables, passed to and from other functions, and stored in data structures (tables, notably). Functions as first-class citizens enable a functional style of PL/SQL programming that (1) nicely complements existing practice but also (2) paves the way for new, particularly concise and elegant query idioms.

The "*functions as data*" idea is effective in the sense that it brings far-reaching query formulation opportunities while few language extensions suffice to anchor the paradigm in PL/SQL:[1]

- `FUNCTION(`t_1`) RETURNS` t_2, the type of functions from t_1 to t_2, is now a data type (just like `INTEGER` or `VARCHAR(`n`)`),
- function names—built-in (like `atan` or `upper`) and user-defined—denote regular values (alas of function type),
- `FUNCTION(`$x\ t_1$`) RETURNS` t_2 `AS BEGIN` e `END` denotes a literal function with argument x and body e (just like `42` denotes a literal of type `INTEGER`), and finally
- e`(`e_1`,...,`e_n`)` is regarded as a valid (dynamic) function call if expression e evaluates to a value of function type.

Further, first-class functions come with a lightweight and efficient implementation approach, *defunctionalization* [8], that does not require database kernel changes [4]. Cast as a source-to-source transformation, defunctionalization can be non-invasively applied to any PL/SQL host. The present demonstration builds on a PL/SQL preprocessor that sits entirely on top of PostgreSQL 9 [1].

2. PL/SQL WITH A FUNCTIONAL MINDSET

Despite being a modest language extension with a non-invasive implementation, first-class functions have a profound impact on how ideas can be expressed in terms of PL/SQL queries. We have collected three sample application scenarios here, ordered from customary to offbeat, and include many more in the actual software demonstration.

In the PL/SQL listings below, we have placed a mark (⋮) in the gutter where the language extensions come into play.

Ⓐ **Functions That Travel With Data.** In a TPC-H benchmark database, the status of an order is determined by the status of its line items [9, § 4.2.3]: if all line items either agree on status `'F'` or `'O'` (finalized *vs.* open) then this also

[1] Of course, the idea does not hinge on this particular syntax.

ORDERS

o_orderkey	o_orderstatus	···
1	'F'	
2	'P'	
⋮	⋮	

LINEITEM

l_orderkey	l_linestatus	···
1	'F'	
1	'F'	
2	'O'	
2	'O'	
2	'F'	
⋮	⋮	

Figure 1: Static status of orders (`o_orderstatus`) and their line items (`l_linestatus`) in a TPC-H instance.

```
1  -- implements order status constraint as per TPC-H §4.2.3
2  CREATE FUNCTION tpch_constraint_423(o ORDERS)
3   RETURNS FUNCTION(CHAR (1)) RETURNS CHAR(1) AS
4  BEGIN
5   RETURN FUNCTION(s CHAR(1)) RETURNS CHAR(1) AS
6          DECLARE status CHAR(1);
7          BEGIN
8           -- query the status of the line items of order o
9           SELECT DISTINCT li.l_linestatus
10          INTO   STRICT status
11          FROM   LINEITEM li
12          WHERE  li.l_orderkey = o.o_orderkey;
13
14          RETURN status;
15
16          EXCEPTION
17           -- when line items disagree on status ...
18           WHEN TOO_MANY_ROWS THEN RETURN s;
19         END;
20 END;
21
22 -- extend table ORDERS with functional column o_livestatus
23 ALTER TABLE ORDERS
24   ADD COLUMN o_livestatus FUNCTION(CHAR(1)) RETURNS CHAR(1);
25 UPDATE ORDERS o SET o_livestatus = tpch_constraint_423(o);
26
27 -- retrieve all orders and their status as of now
28 SELECT o_orderkey,
29        o_livestatus('P')
30 FROM   ORDERS;
```

Figure 2: Higher-order function encodes a TPC-H constraint.

is the order's status. Otherwise the order has status `'P'` (processing). When the TPC-H data generator `DBGEN` populates the instance, it statically sets column `o_orderstatus` of table `ORDERS` accordingly (see Figure 1).

First-class functions can help to implement the above consistency constraint in an alternative, dynamic fashion. To this end, the PL/SQL code of Figure 2 extends table `ORDERS` with function-valued column `o_livestatus` and populates it for all orders `o` (lines 23 to 25): `tpch_constraint_423(o)` returns a function that, when invoked by a query, will perform status computation for order `o`, thus reflecting live updates of its line items. After the update in line 25 has been processed, table `ORDERS` takes the form shown here. We add flexibility in that the functions in new column `o_livestatus` accept the `CHAR(1)` argument `s`, the order status returned should the status of the line items disagree.[2]

ORDERS

o_orderkey	···	o_livestatus
1		FUNCTION(s)
2		FUNCTION(s)
⋮		⋮

Queries reference columns of function type just like first-order columns (see line 29 where a dynamic function call invokes `o_livestatus` to compute the live order status as of query time).

[2] `'P'` would be the customary argument here but, *e.g.*, `NULL` might be appropriate in other contexts.

Note how `tpch_constraint_423(o)` operates like a factory, or higher-order function, that constructs a function tailored to determine the status of its particular order argument `o`: the function literal defined in lines 5 to 19 refers to row variable `o` and its key `o.o_orderkey` to identify the associated line items. Under the hood, defunctionalization captures the value of such free variables at runtime, *i.e.*, when the function is defined, and bundles these values together with (a reference to) the function's body code. When the function is invoked later on, its references to free variables are resolved using the values stored in the bundle (or *closure* [5]). The closures representing the functional values `FUNCTION(s)` in the extended `ORDERS` table above thus take the form ⟨*code lines 5–19* | `o`⟩. We come back to closures and their relational representation below.

Ⓑ Routing Functions and Arguments. When functions reside in tables next to regular values, we can adopt a programming style in which queries may be used to flexibly route arguments to their functions. To make this point, the PL/SQL example code of Figure 3 creates and then populates a table `FUNS` in which column `fn` holds real-valued functions: the built-in and user-defined functions `atan` and `square` are considered data as is the literal doubling function with `id = 3` (line 11). For any such function f in tables `FUNS`, this example aims to tabulate f side by side with its *first derivative* f' (from such a tabulation we can easily derive plots). We exploit that the differential quotient of f approximates f' if the distance parameter h is small:

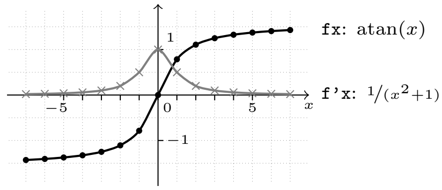
$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for small } h \enspace .$$

```
CREATE FUNCTION square(x REAL) RETURNS REAL AS                        1
BEGIN                                                                 2
 RETURN x * x;                                                        3
END;                                                                  4
                                                                      5
CREATE TABLE FUNS (id INTEGER NOT NULL PRIMARY KEY,                   6
                   fn FUNCTION(REAL) RETURNS REAL);                   7
INSERT INTO FUNS VALUES                                               8
 (1, atan),   -- built-in function                                    9
 (2, square), -- user-defined function                                10
 (3, FUNCTION(x REAL) RETURNS REAL AS BEGIN RETURN 2 * x; END);       11
                                                                      12
-- compute differential quotient for function f                       13
CREATE FUNCTION diffq(h REAL, f FUNCTION(REAL) RETURNS REAL)          14
    RETURNS FUNCTION(REAL) RETURNS REAL AS                            15
BEGIN                                                                 16
 RETURN FUNCTION(x REAL) RETURNS REAL AS                              17
        BEGIN                                                         18
         RETURN (f(x + h) - f(x)) / h;                                19
        END;                                                          20
END;                                                                  21
                                                                      22
-- compute first derivative of function f                             23
CREATE FUNCTION derive(f FUNCTION(REAL) RETURNS REAL)                 24
    RETURNS FUNCTION(REAL) RETURNS REAL AS                            25
BEGIN                                                                 26
 RETURN diffq(0.001, f); -- fix a small h, here: 0.001                27
END;                                                                  28
                                                                      29
CREATE TABLE ARGS (x REAL NOT NULL);                                  30
INSERT INTO ARGS VALUES (-100.0),(-99.0),...,(99.0),(100.0);          31
                                                                      32
-- tabulation of all functions along with their first ;derivatives    33
SELECT id, x,                                                         34
       fn(x) AS fx,                                                   35
       derive(fn)(x) AS "f'x"                                         36
FROM   FUNS, ARGS;                                                    37
```

Figure 3: Tabulating functions along with their derivatives.

id	x	fx	f'x
⋮	⋮	⋮	⋮
1	-1.0	-0.78	0.5
1	0.0	0.00	1.0
1	1.0	0.78	0.5
1	2.0	1.10	0.2
⋮	⋮	⋮	⋮

(a) Tabulation.



(b) Plotting the query result for `atan` (`id` = `1`).

Figure 4: Function `atan` and its first derivative (plotting the result of the SQL query of Figure 3 for `id` = `1`).

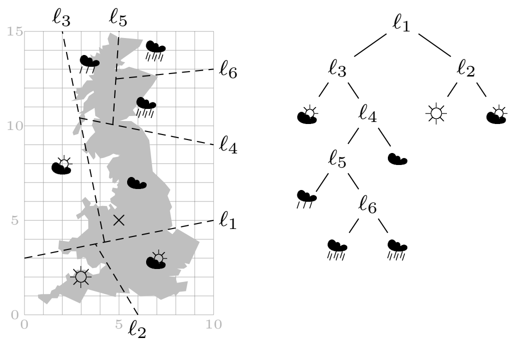


Figure 5: A binary space partitioning tree representing a weather forecast (cloudy spot × is left of ℓ_1 and right of $\ell_{3,4}$).

Here, we understand the derivation operator $\square'$ as being higher-order in that it maps its functional argument f onto the first derivative, *i.e.*, another function. Function `derive` and its auxiliary `diffq` directly embody this understanding: for a given real-valued PL/SQL function f, `derive(`f`)` constructs a new function that approximates the first derivative of f (lines 14 to 28).

To complete the example, we set up a second table `ARGS` of function arguments `x`. The SQL query in lines 34 to 37 then applies the functions in table `FUNS` along with their derivatives to all arguments in table `ARGS` to form the tabulation. Note how, in lines 35 and 36, `fn` (referring to the values in the second column of table `FUNS`) as well `derive(fn)` (a derivative constructed at runtime) denote functions and thus may be applied to the current argument `x`. Figure 4 shows an excerpt of the query result in tabular form as well as the associated plot.

Ⓒ **Algebraic Data Types.** Functional programming is closely linked to *algebraic data types*—tree-shaped data types whose instances are built through the application of functions [5]. The shape of an algebraic data type is typically specified in terms of a recursive equation. Consider (adopting Haskell syntax here, read :: as "*has type*" and | as "*or*")[3]

```
data BSP = Part {left :: BSP, line :: LSEG, right :: BSP}
         | Leaf {label :: TEXT} .
```

Type `BSP` describes binary trees whose inner nodes carry line segments that partition an underlying two-dimensional space into a left and right half; leaves carry a textual label. The data type equation also defines the constructor functions (here: `Part` and `Leaf`) that are used to build trees of this shape. Such binary space partitioning trees might be used to structure a weather forecast map, for example (see Figure 5).

[3]PostgreSQL's built-in type `LSEG` represents line segments.

```
SELECT Part(Part(Leaf(wales),
                 ℓ3,
                 Part(Part(Leaf(westscotland),
                           ℓ5,
                           Part(Leaf(shetlands),ℓ6,Leaf(scotland))),
                      ℓ4,
                      Leaf(midlands))),
            ℓ1,
            Part(Leaf(southwest),ℓ2,Leaf(southeast)))
FROM   UK_FORECAST
WHERE  day = 'tomorrow' :: DATE;
```

Figure 6: Constructor calls build the binary space partitioning tree of Figure 5 to form a regional weather forecast map.

PART

id	left	line	right
α_1	α_3	ℓ_1	α_2
α_2	β_1	ℓ_2	β_2
α_3	β_3	ℓ_3	α_4
α_4	α_5	ℓ_4	β_4
α_5	β_5	ℓ_5	α_6
α_6	β_6	ℓ_6	β_7

LEAF

id	label
β_1	' '
β_2	' '
β_3	' '
β_4	' '
β_5	' '
β_6	' '
β_7	' '

Figure 7: Tabular closure storage. The keys α_i, β_j serve as the closure representation.[4]

Church [3] made the key observation that first-class functions suffice to encode any algebraic data type—we need no special provisioning to use these expressive types in PL/SQL programs. In the Church encoding, constructors return recursive functions, *folds* [7], that can be used to traverse the built instance. The tree itself remains implicit. We show the `Part` constructor below (`Leaf` is defined analogously):[4]

```
1 CREATE FUNCTION Part(left BSP,line LSEG,right BSP) RETURNS BSP AS
2 BEGIN
3  RETURN FUNCTION(l FUNCTION(TEXT) RETURNS t,
4                  p FUNCTION(t, LSEG, t) RETURNS t) AS
5         BEGIN
6          RETURN p(left(l, p), line, right(l, p));
7         END
8 END;
```

Once the constructors are in place, they may be conveniently used in SQL queries: the query of Figure 6 builds a two-dimensional map from flat weather forecast data. A `lookup` function of type `FUNCTION(BSP,POINT) RETURNS TEXT` ("*how is the weather in spot* ×*?*") can be straightforwardly defined.

As mentioned before, defunctionalization trades functions for closures that bundle a code reference plus the function's environment of free variables. The function returned by constructor `Part` above turns into ⟨*code lines 3–7* | `left`, `line`, `right`⟩, for example. Whenever these bundles nest—as is the case here: free variables `left` and `right` are bound to functions, and thus closures—we have designed defunctionalization to (1) save closures into tables and (2) use the tables' key to serve as the closure representation instead. Figure 7 depicts the closure tables that result from the `Part` and `Leaf` calls performed by the query of Figure 6. Note how the nested constructor invocations in the defunctionalized code *implicitly* built a relational representation of the binary space partitioning tree. First-class PL/SQL functions have introduced an abstraction that saves the developer from explicitly wiring the tree's nodes.

[4]We omit the PL/SQL definition of type `BSP` here. It may, just like the constructor definitions, be mechanically derived from the equation of the algebraic data type.

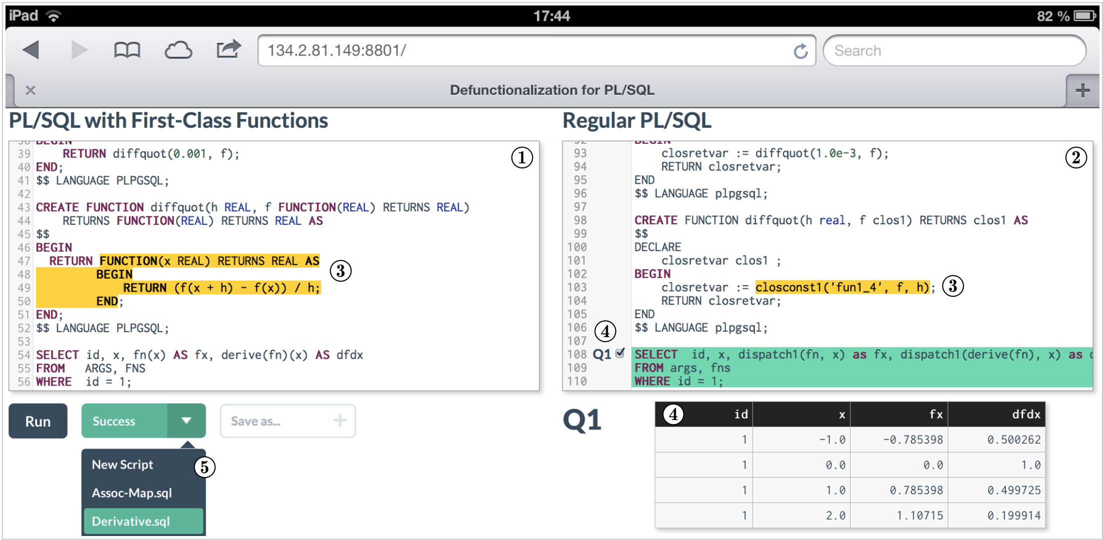


Figure 8: Screenshot of the web-based demonstrator. Code highlights at ③ illustrate how defunctionalization translates a literal function into a closure constructor (here: `closconst1()` bundling code symbol `fun1_4` with free variables `f` and `h`).

3. DEMONSTRATION SETUP

"*Functions as data*" not only characterizes the class of query idioms that is in our toolbox now, but also hints at the implementation technique used in this work. Query defunctionalization [4] trades functional values for regular first-order data items which off-the-shelf relational DBMSs process efficiently. This translation from source program with first-class functions into regular PL/SQL target code is reflected by the demonstrator's screen layout and operation (see Figure 8). Users compose PL/SQL input in editor window ①—the demonstrator responds with an equivalent runnable program in output window ②.

In a nutshell, a function's closure $\langle code \mid v_1, \ldots, v_n \rangle$ turns into (a) a symbol that stands in for the *code*, plus (b) an entry into the table that saves the bindings of the function's free variables $v_1, \ldots, v_n$ (recall tables `PART` and `LEAF` of Figure 7). Under the defunctionalization transformation, some source language constructs may affect *multiple* spots of the generated target program. An occurrence of a function literal like `FUNCTION(`$x\ t_1$`) RETURNS` t_2 `BEGIN` e `END`, for example,

(1) is replaced by a constructor that introduces a code symbol for e and builds the required closure, then
(2) creates a regular (top-level, named) function that wraps the literal's body statement sequence e, and
(3) generates an auxiliary PL/SQL routine that dispatches to the wrapper where the source program would invoke the function literal.

Dynamic function calls and named function references are subject to analogous translations.

The demonstration illustrates this correspondence between source and target constructs through interactive code highlighting (see ③ in Figure 8). These highlights track cursor movement with fine granularity—at the level of individual statements and expressions—and thus help to quickly develop a solid intuition of the ideas behind defunctionalization.

The web-based demonstrator client UI is connected to a PostgreSQL instance and target programs may be executed directly within the environment. Checkboxes are placed in the output window's gutter such that the results of SQL DML statements may be selectively shown or hidden (in the screenshot, at ④ we have chosen to render the result of SQL query *Q1*).

Beyond the use cases sketched in Section 2, we have preloaded the system (⑤) with a wide range of application scenarios to demonstrate the gains that come with first-class PL/SQL functions. Users will find samples of, for example,

- a variant of associative maps, or key/value dictionaries, that elegantly model inter-table references even if these span multiple relations or involve more than plain foreign-key joins,
- functions that naturally add flexibility to otherwise rather static database schemata (*e.g.*, pricing schemes for TPC-H orders that are configurable on a per-tuple basis), and
- combinators that capture intricate query patterns in a concise fashion (*e.g.*, fixpoint computations that find a graph's connected components as recently described in [6]).

Acknowledgments. This research is supported by the German Research Council (DFG) under grant no. GR 2036/3-2.

4. REFERENCES

[1] PostgreSQL 9.2. `www.postgresql.org/docs/9.2/`.
[2] *Oracle Database PL/SQL Language Reference—11g Release 1 (11.1)*, 2009.
[3] A. Church. The Calculi of Lambda-Conversion. *Annals of Mathematics Studies*, volume 6, 1941.
[4] T. Grust and A. Ulrich. First-Class Functions for First-Order Database Engines. In *Proc. DBPL*, 2013.
[5] P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.
[6] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *Proc. CIDR*, 2013.
[7] T. Sheard and L. Fegaras. A Fold for All Seasons. In *Proc. FPCA*, 1993.
[8] A. Tolmach and D. Oliva. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *J. Funct. Programming*, 8(4), 1998.
[9] Transaction Processing Performance Council. *TPC-H, a Decision-Support Benchmark*. `tpc.org/tpch/`.