FERRY — Database–Supported Program Execution

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber

WSI, Universität Tübingen
Tübingen, Germany

⟨*firstname.lastname*⟩@uni-tuebingen.de

ABSTRACT

We demonstrate the language FERRY and its editing, compilation, and execution environment FERRYDECK. FERRY's type system and operations match those of scripting or programming languages; its compiler has been designed to emit (bundles of) compliant and efficient SQL:1999 statements. FERRY acts as glue that permits a programming style in which developers access database tables using their programming language's own syntax and idioms—the FERRY-expressible fragments of such programs may be executed by a relational database back-end, *i.e.*, close to the data. The demonstrator FERRYDECK implements *compile-and-execute-as-you-type* interactivity for FERRY and offers a variety of (graphical) hooks to explore and inspect this approach to database-supported program execution.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query Languages, Database (persistent) programming languages*

General Terms: Languages, Performance

Keywords: Ferry, Pathfinder, LINQ, SQL:1999

1. DATABASE–TROUBLED *VS.* DATABASE–SUPPORTED PROGRAMS

Friday, late afternoon. "*Oh, and we need your program to extract the two top-paid employees and their salaries per department.*" Does this request spoil your weekend plans? Your answer may also depend on whether you are developing in a *database-troubled* or a *database-supported* programming environment:

Ⓐ **Database-troubled: ODBC-style database access.** The request suggests the construction of a data structure in which each department is associated with the list of its two top-paid employees. Figure 1 shows the source database table Employees. While the host language (here: C) supports such data nesting, the underlying 1NF relational database does not. Your ODBC-based program (Figure 2) thus embeds two SQL query strings defining (a) an outer query `depts` (line 8) over whose re-

Employees			
id	name	dept	salary
1	Alex	DE	300
2	Bert	DE	100
3	Cora	DE	200
4	Drew	US	200
5	Erik	US	400
6	Fred	US	300
7	Gina	US	200
8	Herb	NL	600
9	Ivan	NL	400
10	Jill	UK	500

Figure 1: Table Employees.

```
1  ⟨...25 lines of code suppressed...⟩
2  SQLPrepare(emps, "\
3  ␣␣SELECT␣name,salary␣FROM␣Employees␣WHERE␣dept␣=␣?␣\
4  ␣␣ORDER␣BY␣salary␣DESC␣FETCH␣FIRST␣2␣ROWS␣ONLY", ...);
5  SQLBindCol(emps, 1, ..., name, 4+1, ...);
6  SQLBindCol(emps, 2, ..., &salary, 0, ...);
7  SQLBindParameter(emps, 1, SQL_PARAM_INPUT, ..., dept, ...);
8  SQLExecDirect(depts, "SELECT␣DISTINCT␣dept␣FROM␣Employees", ...);
9  SQLBindCol(depts, 1, ..., dept, 0, ...);
10
11 rc = SQLFetch(depts);
12 while (rc != SQL_NO_DATA_FOUND) {
13   SQLExecute(emps);
14   rc = SQLFetch(emps);
15   while (rc != SQL_NO_DATA_FOUND) {
16     ⟨process bound variables dept, name, salary here⟩
17     rc = SQLFetch(emps);
18   }
19   SQLCloseCursor(emps);
20   rc = SQLFetch(depts);
21 }
```

Figure 2: ODBC code excerpt (host language: C).

sults you iterate to obtain bindings for column `dept`, and (b) an inner query `emps` (line 2) that uses these bindings to retrieve columns `name`, `salary` for the two top-paid employees in the respective department.

In effect, the two nested `while` loops of your program take on parts of the query processor's job, submitting the inner query once for each distinct value in column `dept`, *i.e.*, four times. Code length approaches 100 lines (Figure 2 suppresses ODBC connection management, handle allocation, and also omits error handling). You have used a significant subset of the ODBC API (9 different functions appearing in 17 call sites). Tedious plumbing dominates and you still end up with fragile, insidious code (note how outer and inner query use the stack address of C variable `dept` to communicate bindings, lines 7 and 9) that cannot be fully checked at compile time due to ODBC's reliance on query strings.

Ⓑ **Database-supported program execution.** Your script (here: RUBY, Figure 3) reads list `Employees` whose contents reflects the rows of table Employees of Figure 1 [1]. RUBY method `group_by` groups the employees by department and variable `d_es` is bound to a nested list of shape `[(dept,[(id, dept,name,salary)])]`. You use two nested RUBY `map` invocations (line 3) to project on `name`, `salary` in the inner lists and then sort the latter by descending salary before you trim to only retain the first two elements (`sort_by`, `first` in line 4).

Your program largely relies on built-in RUBY functions whose semantics may be understood in terms of disciplined iteration (or comprehensions [7]). You thus know that your programming environment will translate a large fragment

```
1 ⟨objects in Employees have methods id, name, dept, salary⟩
2 d_es = Employees.group_by {|e| e.dept}
3 d_ns = d_es.map {|d,es| [d, es.map {|e| [e.name, e.salary]}]}
4 top2 = d_ns.map {|d,ns| [d, ns.sort_by {|n,s| -s}.first(2)]}
5 ⟨result in top2 (list of shape [(dept,[(name,salary)])])⟩
```

Figure 3: Ruby script.

```
let e = table Employees (id int, name string,
                         dept string, salary int)
      with keys ((id))
 in for x in e
    group by x.dept
    return ⌜(the (x.dept),                                  ⌝
             take (2, for y in zip (x.name, x.salary)
                      order by y.2 descending
            ⌞         return y))                           ⌟
```

Figure 4: Ferry sample Program P_1, to be evaluated against the database table Employees of Figure 1.

of your code into FERRY, a language that—like RUBY—operates over nested, ordered lists but can be compiled into (bundles of) SQL:1999 statements. This program fragment will be executed by the database back-end, close to the Employees table, releasing the RUBY runtime from this data processing task. Weekend saved.

2. THE FERRY GLUE LANGUAGE

FERRY has been designed as an intermediate language onto which a variety of front-end scripting or programming languages can be mapped. FERRY's type system, operations, and function library support computation, in particular iteration, over arbitrarily nested, ordered lists and tuples. All data manipulated by FERRY programs is shaped according to the recursive type equation $t = a \mid [t] \mid (t, \ldots, t)$ where a represents atomic types like *string*, *int*, or *bool*. These types t model programming language types such as lists, (associative) arrays, or dictionaries (the FERRY type $[(int, [string])]$ can represent a string hash with integer keys, for example).

FERRY features a fully orthogonal expression-oriented syntax that revolves around the `for`-`where`-`group by`-`order by`-`return` construct. Figure 4 shows the FERRY sample Program P_1 which realizes the semantics of the RUBY script fragment of Figure 3. The principal comprehension expression `for` x `in` e_1 `return` e_2 performs the side-effect-free iterated evaluation of expression e_2 under successive bindings (supplied by e_1) of variable x. FERRY comes equipped with a library of built-in functions (Figure 5) that has been inspired by HASKELL's standard prelude [5]. A typical FERRY program chains functions that consume and produce lists to realize more complex functionality.

The language design builds on FERRY's nested list types to provide a natural and powerful form of grouping [7]: in the scope of the `group by` clause (marked by ⌜⌝⌞⌟ in Figure 4), expression `x.salary` is of type $[int]$, representing *all* salaries in the current group. The result of Program P_1 has type $[(string, [(string, int)])]$ and reads

```
[ ("DE", [("Alex", 300), ("Cora", 200)]),
  ("US", [("Erik", 400), ("Fred", 300)]),
  ("NL", [("Herb", 600), ("Ivan", 400)]),
  ("UK", [("Jill", 500)]) ] .
```

This nested value is mapped back onto the RUBY heap to be bound to variable `top2` (Figure 3, line 4).

The resemblance of FERRY's `for`-`where`-`group by`-`order by`-`return` block with XQUERY's FLWOR syntax is not accidental as both languages are built on a common semantic ground: *list comprehensions* [7]. Comprehensions are found, under varying coats of syntactic sugar, in a growing family of languages (*e.g.*, C# 3.0, ERLANG, HASKELL, PERL 6, PYTHON, SCALA, RUBY). Indeed, a comprehension-based core is characteristic of the languages that may be understood in terms of FERRY and thus executed with database support.

`map` :: $(t \to t_1, [t]) \to [t_1]$	map over list
`concat` :: $[[t]] \to [t]$	list flattening
`take`, `drop` :: $(int, [t]) \to [t]$	keep/remove list prefix
`nth` :: $(int, [t]) \to t$	positional list access
`zip` :: $([t_1], .., [t_n]) \to [(t_1, .., t_n)]$	n-way positional
`unzip` :: $[(t_1, .., t_n)] \to ([t_1], .., [t_n])$	merge and split
`unordered` :: $[t] \to [t]$	disregard list order
`length` :: $[t] \to int$	list length
`all`, `any` :: $[bool] \to bool$	quantification
`sum`, `min`, `max` :: $[a] \to a$	list aggregation
`the` :: $[t] \to t$	group representative
`groupWith` :: $(t \to (a_1, .., a_m), [t]) \to [[t]]$	grouping (as in [7])

Figure 5: Sample of Ferry's built-in function library.

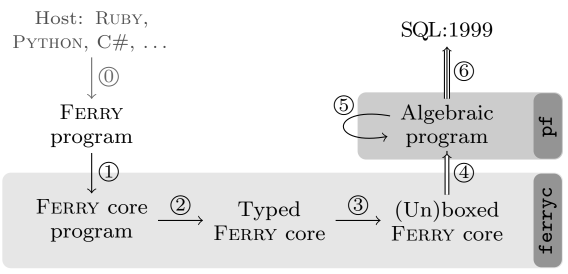


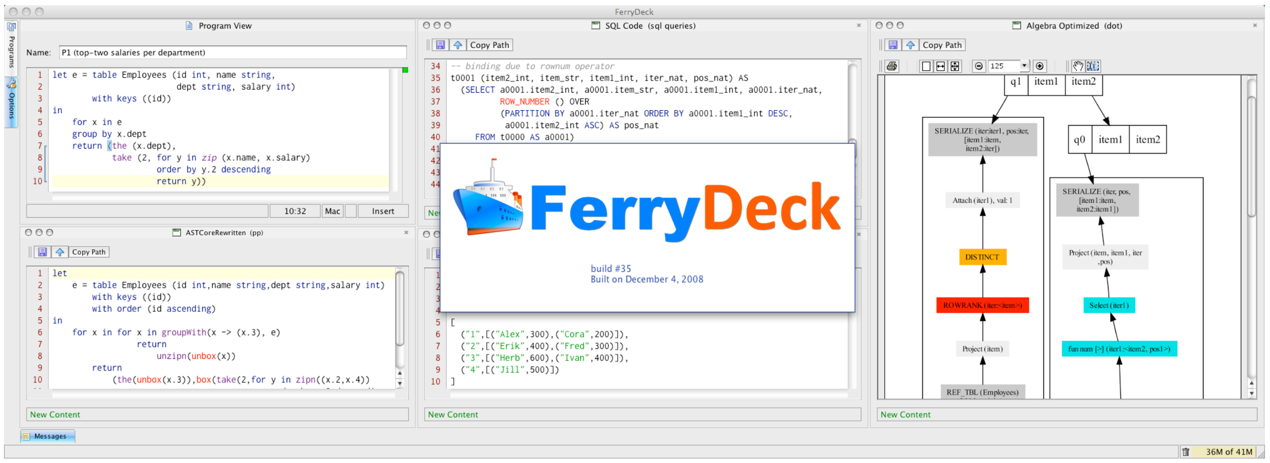
Figure 6: Ferry compilation stages and intermediate program forms.

Compiling Ferry to SQL. To derive database-executable code from a FERRY program, FERRY's compiler `ferryc` applies an adaptation of the *loop lifting* translation strategy [2]. Loop lifting, originally developed for the purely relational PATHFINDER XQUERY compiler, yields compact algebraic query plans for input programs that feature ordered iteration, conditionals, variable bindings and reference, *etc.* Loop-lifted algebraic plans facilitate data-flow-based analysis and optimization and rely on a particularly simple table algebra variant, designed to model the query engine capabilities of modern RDBMS.

Figure 6 sketches the resulting compilation pipeline. In stages ④ and ⑤, `ferryc` employs PATHFINDER's optimizer and code generator `pf` [2] to translate algebraic plans into SQL statement bundles ready for execution by the back-end.

3. STEERING THE FERRY (DEMO SETUP)

FERRYDECK implements a fully interactive editing, compilation, execution, and inspection environment for FERRY (Figure 7 shows a screen shot of FERRYDECK taken while we experimented with Program P_1). The demonstrator FERRYDECK hooks into (a) stages ① to ⑥ of Figure 6, (b) the shipment of SQL:1999 code for execution by the back-end (IBM DB2 V9.5 in this case), and (c) the assembly of the final result to be mapped back onto the programming language heap. Views may be flexibly rearranged to peek at the various intermediate forms a compiled FERRY program takes on. Live syntax checks and updates of *all views*, including the database-computed query result, are performed while



```
let e = table Employees (id int, name string,
                         dept string, salary int)
        with keys ((id))
in
    for x in e
    group by x.dept
    return (the (x.dept),
            take (2, for y in zip (x.name, x.salary)
                          order by y.2 descending
                          return y))
```

```
let
    e = table Employees (id int,name string,dept string,salary int)
        with keys ((id))
        with order (id ascending)
in
    for x in for x in groupWith(x -> (x.3), e)
                 return
                     unzipn(unbox(x))
        return
           (the(unbox(x.3)),box(take(2,for y in zipn((x.2,x.4))
```

```
-- binding due to rownum operator
t0001 (item2_int, item_str, item1_int, iter_nat, pos_nat) AS
  (SELECT a0001.item2_int, a0001.item_str, a0001.item1_int, a0001.iter_nat,
          ROW_NUMBER () OVER
          (PARTITION BY a0001.iter_nat ORDER BY a0001.item1_int DESC,
           a0001.item2_int ASC) AS pos_nat
     FROM t0000 AS a0001)
```

```
[
  ("1",[("Alex",300),("Cora",200)]),
  ("2",[("Erik",400),("Fred",300)]),
  ("3",[("Herb",600),("Ivan",400)]),
  ("4",[("Jill",500)])
]
```

q1
item1
item2
SERIALIZE (iter:iter1, pos:iter, [item1:item, item2:iter])
Attach (iter1), val: 1
DISTINCT
ROWRANK (iter:<item>)
Project (item)
REF_TBL (Employees)
q0
item1
item2
SERIALIZE (iter, pos, [item1:item, item2:item1])
Project (item, item1, iter ,pos)
Select (iter1)
fun num [>] (iter1:<item2, pos1>)

Figure 7: Screen shot of FerryDeck taken while processing P_1. As you type, FerryDeck translates the input program and continuously updates views of the compiled program, plan bundle, emitted SQL code, and result.

users edit their FERRY programs. With loop lifting, `ferryc` relies on a compositional compilation strategy that does not break when facing large or deeply nested programs. (This submission is accompanied by a three-minute screencast.)

Visualizing plan bundles. To implement its nested data model on 1NF database back-ends, FERRY represents nested lists by *surrogate keys* (see surrogate 2, column box, Figure 8) [6]. For each list constructor $[t]$ occurring in a program's result type, FERRY generates a separate algebraic plan computing the partial result at that nesting level. This dependency on *type* is in contrast to Scenario Ⓐ of Section 1 where the *data* determined the number of issued queries. The programming language runtime benefits from this plan bundling as it can consume FERRY results gradually at different levels or partially only. FERRYDECK users can easily experience the effects if they let their program "open boxes" (*i.e.*, inspect a nested list's actual elements) selectively. FERRYDECK performs prompt PDF re-rendering of plan bundles (Figure 7, right) and users see bundles dynamically grow or shrink as they alter the nesting expressed by their programs.

Q_0

iter	pos	dept	box
1	1	DE	1
1	2	US	2
1	3	NL	3
1	4	UK	4

Q_1

iter	pos	name	salary
1	1	Alex	300
1	2	Cora	200
2	1	Erik	400
2	2	Fred	300
3	1	Herb	600
3	2	Ivan	400
4	1	Jill	500

Figure 8: Table pair representing Program P_1's nested result.

Preserving order on an unordered back-end. Loop lifting embeds information on iteration and list order *in the data* (columns iter, pos, Figure 8) [2]. While this explicit encoding of order does incur runtime cost, the compiler is provided with effective handles to selectively enforce or omit order maintenance: if FERRYDECK users edit a program to locally ignore order, *e.g.*, via the introduction of a call to built-in function `unordered`(·) (Figure 5), they can immediately notice a global reduction of plan complexity.

SQL:1999 code generation and execution. FERRY's code generator `pf` turns an algebraic plan into a bundle of SQL:1999 statements that collectively realize the plan's semantics. FERRYDECK lets users browse the statement groups as well as their individual tabular results (cf. Q_0, Q_1 of Figure 8) as computed by IBM DB2 V9.5. Additionally, the demonstrator merges the tables to provide the host language's view of a potentially deeply nested value.

Ferry and Linq. FERRY and its library of built-in functions embrace the *Standard Query Operators* of Microsoft's LINQ [4], another member of the family of comprehension-based languages. With its SQL code generator, FERRY makes for an ideal option to realize an alternative, non-proprietary LINQ-to-SQL provider. FERRY leaps ahead with data-flow-based join detection that helps to avoid the iterative evaluation of nested database queries. Each such iteration entails an execution context switch from programming language runtime to database query processor and back—a phenomenon found in LINQ execution traces. Microsoft's current LINQ-to-SQL provider organizes and accesses the elements of nested lists by table offsets (here, LINQ relies on specific Microsoft SQL Server behavior that can guarantee ordered row storage and permits positional access). FERRY's treatment of data nesting is not tied to characteristics of a particular database back-end and thus brings any SQL:1999 RDBMS within reach of the LINQ technology. The demonstration features FERRYDECK and a LINQ GUI (LINQPAD [3]) side by side to make the opportunities in this space tangible.

Acknowledgments. This research has been supported by the German Research Association (DFG), Grant GR 2036/1.

4. REFERENCES

[1] ACTIVERECORD. `http://ar.rubyonrails.org`.
[2] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.
[3] LINQPAD. `http://www.linqpad.net/`.
[4] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. SIGMOD*, 2006.
[5] S. Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1), 2003.
[6] H.J. Schek and M.H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2), 1986.
[7] P. Wadler and S. Peyton Jones. Comprehensive Comprehensions: Comprehensions with "Order by" and "Group by". In *Proc. ICFP Haskell Workshop*, 2007.