An ODBC Interface for Objective Caml

R. Castro, X.M. López, V.M. Gulías

Abstract

In this paper, an integration of the functional language Objective Caml and the standard API for accesing to relational databases ODBC is presented. By using this interface, the functional programmer can store values in a non-volatile storage in order to recover them in future sessions using a client-server approach. In our proposal, queries to the relational manager are encapsulated into structures quite familiar to the functional programmer, such as lists or tuples. Hence, functional programming is used to manipulate data retrieved from persistent storage while efficient and mature data access is performed using a relational database.

Keywords: Functional Programming, Relational Databases, Client-Server architecture, Integration

1 Introduction

In this paper, an integration of the functional language Objective Caml [7] and the standard API for accesing to relational databases ODBC [5, 13, 6] is presented. The proposed goal is to provide a mechanism to save functional values in persistent storage, using as storage a relational database. Despite the fact that database access or even persistence inside a functional language is not new [12, 10, 9, 8, 4, 11], research in this field has been shadowed by other state-of-the-art subjects, such as semantics or efficient functional code execution.

In our approach, queries to the relational manager are encapsulated into structures quite familiar to the functional programmer, such as lists or tuples. A relationship between these concepts and those of the relational model must be established. As far as we can, remote access to data is hidden by using functional programming idioms that are translated into queries to a relational server (client/server approach). Hence, functional programming is used to manipulate data retrieved from persistent storage while efficient and mature data access is performed using a relational database.

The paper is structured as follows: the next section presents a brief introduction to the standard ODBC interface. Section 3 deals with the integration methodology used. Section 4 presents some useful primitives and how they can be used to implement functional programs that perform database queries. Section 5 shows some benchmarks that measure the overhead introduced by our interface. Section 6 presents a brief discussion on persistence in functional languages. Finally, we conclude and present some future work.

Authors are with the Lfcia, Department of Computer Science, University La Coruña, SPAIN
fax: (+34) (81) 167160. e-mail: {mon,xesus,gulias}@dc.fi.udc.es
Work partially supported by Xunta de Galicia XUGA10504B96

2 The ODBC Interface

In the current database market, there are many different database managers and database formats. It is quite important that all these formats and protocols can be understood by program applications in order to access this information.

In small systems, applications are designed to access databases through a unique DBMS, and hence, the usage of the permanent storage is straightforward. However, large real-world applications often demand data from different sources: multiple databases, multiple DBMS's, different locations, etc., which can increase dramatically the development and maintainance costs of a program. In this setting, a program has to host different APIs (*application program interfaces*) for every DBMS that is (or may be) used.

As an alternative, a common interface has been proposed. This standard API, named ODBC (*Open DataBase Connectivity*) allows us to create applications that can access heterogeneous relational databases using the well-known client-server approach. Now, the programmer does not care about the actual DBMS that is going to be used by the final user.

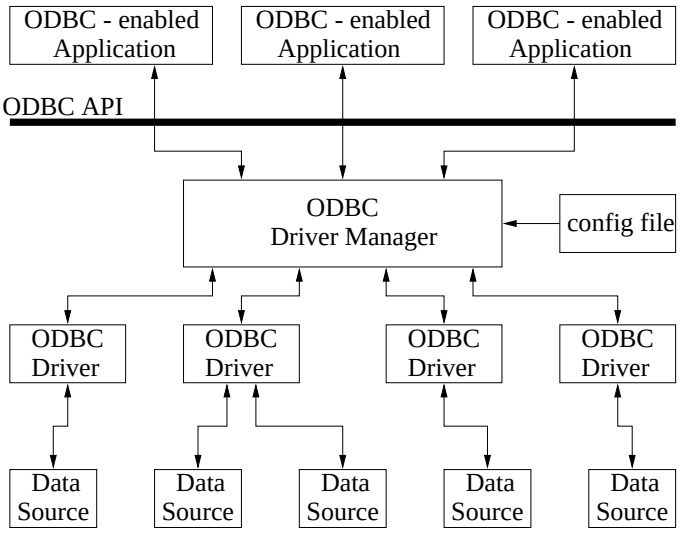

Figure 1: ODBC Architecture

Some of the advantages of using ODBC can be summarized as follows:

- *Multidatabase and multiDBMS access and the interconnection across different platforms.*
- *Portable.* Integration of any application with any DBMS, which allows new programs to use existent databases.
- *Database transparency.* The programmer is isolated from DBMS details.
- *Location transparency.* There is no need for knowing where data is located.
- *Simplify development.* The programmer only needs to know the ODBC API.
- *Performance.* Even though a new layer is added between program and data, ODBC is designed to produce applications with performance similar to native DBMS calls.

The ODBC architecture (figure 1) is composed by four basic components:

- *The actual application.* In order to use the database, it carries out calls to the ODBC API, which generate SQL sentences and finally deliver their results.
- *The driver manager.* Its main goal is to coordinate the load of ODBC drivers to access the associated DBMS on a demand basis. The application uses the ODBC functions through the interface defined by the driver manager, which dynamically loads the appropriate ODBC driver and sends the calls coming from the application to the actual driver. The configuration of the drivers is hold in a separate file in order to maintain independent the application from the DBMS's. The following example shows a fragment of an ODBC configuration file that specifies the driver for a database managed by MySQL [1] (in our tests, we have used MySQL as DBMS).

```
[sampledb]
Driver = /usr/local/lib/libmyodbc_mysql.so
DSN    = sampledb
SERVER = picoro.dc.fi.udc.es
UID    = mon
PWD    =
```

  where `sampledb` is the database identifier which is intended to be accessed using the driver for MySql. `DSN` is the data source name, the `server` is the host where the database is located, and the `UID` and `PWD` are the user and password to the database, respectively.

- *The specific DBMS driver.* It takes the ODBC calls and traslates them to the format expected by the specific DBMS which the driver is designed for, and then translates the results to the standard ODBC format.
- *The data source.* Data which the applications need to access, in addition to the DBMS, the operating system and the communication network.

3 Integration Methodology

In order to integrate ODBC access into a functional language, we use the same methodology proposed in [3]. In that work, our target was to integrate Postgres95 [14] access to the O'Caml functional language using a multi-layer approach. Even though we were successful in this approach, one of the main drawbacks was that we were not able to use comercial databases nor heterogeneous database computing. With the ODBC interface, we expect to bridge this gap.

Figure 2 shows the architecture of the proposed integration.

3.1 Low-level Interaction between ODBC and O'Caml

At the first stage, a consistent interface between ODBC and O'Caml should be provided. An interface library is composed of a set of functions that can be used from another programming language. A common interface between different products consists of using

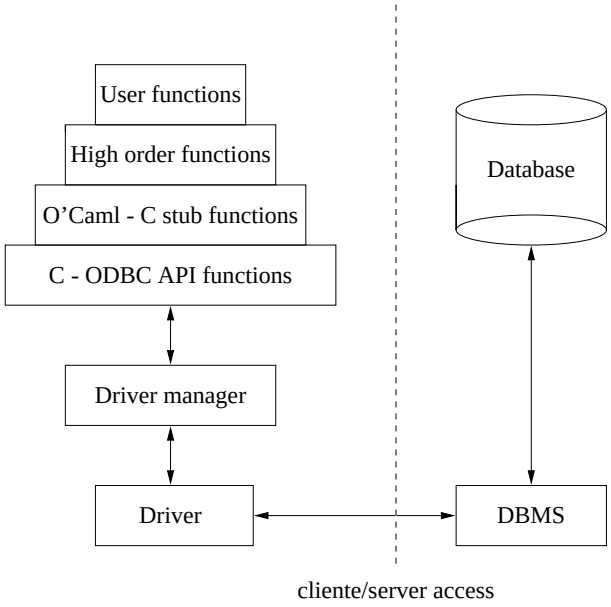

Figure 2: Components of the ODBC-Caml interface

a language like C. Both ODBC and O'Caml provides means to interact with other software applications using the C language.

ODBC provides the driver manager, a static-linked C library that allows, among other things, to establish a connection with the server and to carry out SQL requests. On the other hand, O'Caml incorporates a C interface that allows to define C functions and to use them in the same way as functions defined inside O'Caml.

Of course, things are not so easy. Data structures manipulated by different software packages differ, so conversions must be carried out to exchange information. ODBC establishes *a cursor* as a way of retrieving the result of a query from C. All the arguments and results of the ODBC functions have C types (`int`, `char *`, pointers to low-level structures). Giving that these types are different to that provided by the functional language used in this experiment, suitable conversions must be performed.

3.1.1 Function Encapsulation

Every ODBC function needed by upper layers is encapsulated within a block of code which performs suitable conversions from O'Caml arguments to ODBC arguments, calls the actual entry point of the library function, and then performs a conversion of the result back to O'Caml in order to be used in the functional world (figure 3).

3.1.2 Mapping C Structures to O'Caml Types

Conversion between basic types of both worlds is almost straightforward. There are basic conversion functions provided by the O'Caml C interface. However translations between more complex types, like those that require dynamic memory management, would require a bit more of attention. Dynamic structures must be migrated from O'Caml heap, which is automatically managed by O'Caml runtime system using a garbage collector to reclaim the unused cells, to C heap which is manually managed by the programmer.

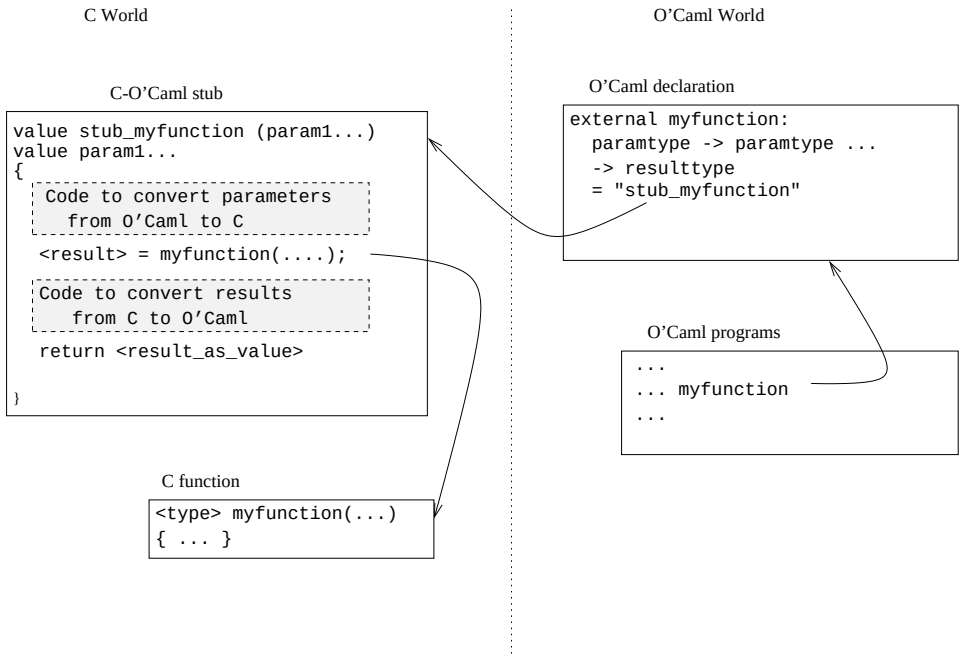

Figure 3: Encapsulation of C functions to be used by O'Caml

C manipulation of ODBC capabilities is quite messy because the programmer must deal with too much detail. For example, two kinds of pointers are allowed in C: *generic pointers*, which can reference any kind of structure in memory, and *concrete pointers*, which only can reference one data type. Moreover, being C a weakly typed language, it permits the conversion between generic and concrete pointers freely. On the other hand, O'Caml includes a variant of Hindley-Milner's polymorphic type inference system, that becomes helpful to the programmer giving that it performs a compile-time type checking avoiding the execution of ill-typed programs.

To simplify programming with the low-level interface, we are going to map generic pointers, used in C to handle different DB objects, to *different* types in O'Caml world, so compile-time type checking will detect incorrect uses of DB objects. For example, it will not be possible to mismatch a connection handler with a query-result handler.

In the ODBC interface there are functions that receive or return pointers to structures such as the aforementioned handlers. For an O'Caml programmer that wants to use a DB, it is not necessary to know the actual values stored in the structures associated with those handlers, because such information is retrieved using additional methods. Given that it is not necessary for O'Caml to know the value of any handler directly, they can be represented using *abstract data types*. For example, a `SQLHENV` pointer (ODBC environment handler) is hidden into the `hEnv` ADT:

```
struct SQLHENV* ⇒ type hEnv;;
```

4 ODBC-O'Caml Interface Basics

4.1 Mapping RDB concepts to the FL Framework

The basic interface defined in section 3.1 establishes a direct link between ODBC and O'Caml. In this second stage, many improvements are implemented, being the main aim of this layer to establish a correspondence between the objects handled by the relational manager and the objects handled by the functional language. In addition, some simplifications are introduced such as default values for the most common arguments. The obvious relationship between both worlds is to think of relational tuples as functional tuples, and relations as lists of functional values. Lists are the most used data structures in functional languages and compilers incorporate special syntactic sugaring as well as library functions for manipulating them.

relational model tuple	$\Rightarrow$	functional model tuple
table or relation	$\Rightarrow$	list

The most important result of this mapping is the *supression of the notion of cursor*. Database acceses retrieve all the information in a list, and that list can be processed with the powerful facilities of the language.

4.2 The ODBC Primitives

A set of higher-level functions have been defined in order to combine the ODBC access with the powerful features of a functional language (table 1). These more abstract definitions isolate many of the tasks associated with the direct O'Caml-C interface and the ODBC's API. They provide the common functionality provided by the data manipulation languages (*DML*'s).

`dbOpen`	`dbDatabase -> dbConnection`
`dbClose`	`dbConnection -> unit`

`dbSelect`	`dbConnection -> dbField list -> dbTable list ->`
	`dbCondition list list ->`
	`dbType -> dbType list`

`dbInsert`	`dbConnection -> dbTable -> dbField list -> dbValue list ->` `dbType -> unit`
`dbUpdate`	`dbConnection -> dbTable -> dbField list -> dbValue list ->` `dbCondition list list -> dbType -> unit`
`dbDelete`	`dbConnection -> dbTable -> dbCondition list list -> unit`

Table 1: High-level Interface

The first function needed is `dbOpen`, which given a valid database name, it delivers an abstract data type `dbConnection`. This ADT represents a connection between the host, the computer which carries out the functional program, and the DBMS, which manages the desired database. As said, the programmer only have to specify the database name; the ODBC driver manager is in charge of looking for more information in the ODBC configuration file, for instance where is located the database. The abstract datatype

ident	name	qualif	dept
365	Gary Baldi	17.3	acc
45	Pepe Perez	13.0	exp
9516	Kurt Schfn	14.3	mar
84	Al Fonso	24.6	cus
47	Filemon Pi	21.5	exp
34	Gong Li	19.1	cus
43	Fran Cisco	19.5	acc
49	Paco Feixo	16.2	sal
3	John Kipur	18.3	imp
53	Bill Bones	11.2	sal

(a) `people` table

ident	name	place
acc	Account	Sebatopol
cus	Customers	Trondheim
sal	Sales	Kinshasa
imp	Imports	Setubal
exp	Exports	Kobe
mar	Marketing	Managua

(b) `depts` table

Figure 4: Database `sampledb`

`dbConnection` isolates the programmer from many details that are only needed at lower layers.

`dbSelect` retrieves tuples from the database once the connection has been established. Each argument of the function represents the fields (`dbField list`), tables (`dbTable list`), and conditions (`dbCondition list list`) of a SQL statement. The meaning of the condition list is as follows:

$$[[c_{11};\ldots;c_{1n}];\ldots;[c_{m1};\ldots;c_{mn}]] \equiv (c_{11} \vee \ldots \vee c_{1n}) \wedge \ldots \wedge (c_{m1} \vee \ldots \vee c_{mn})$$

The rest of the functions, `dbUpdate`, `dbInsert`, and `dbDelete`, update, insert, and delete data from the database, respectively. The exception `DbError` has been defined in order to capture error conditions during the interaction between O'Caml programs and the ODBC interface. The exception is parametrized with an string that indicates the error that has been detected.

4.3 Dynamic and Static Typing

We use O'Caml static type system as much as possible. However, it is necessary to introduce some dynamic runtime type checks in order to assure the correct interaction between O'Caml and ODBC. The `dbType` annotation is just an example of the type expected for every tuple in the result list. In order to make things easier, we have defined some constants with the same names as the basic types. For instance, the annotation `(int,float,string)` is a value of type `(int * float * string)`. The goal of the `dbType` annotation is two-fold. At compile time, the `dbType` annotation is used to force that the values taken from the database match the proposed signature and, hence, they can be freely manipulated by the rest of O'Caml.

Figure 4 shows a database (`sampledb`) with two tables (`people` and `depts`). The following example shows a program which is rejected at compile time by the type checker:

```
# let q' = dbSelect db ["name"] ["people"] [["qualif>=20"]] (string)
  in
     List.hd q' + 3;;
```

```
This expression has type string but is
  here used with type int
```

The `dbType` annotation forces that local definition `q'` to have type `string list`. `List.hd` takes the first element (if any) of the result list `q'`, so `List.hd q'` must have type `string`: Type error raises because we are trying to add a string to an integer with the predefined operator `(+) : int -> int -> int`.

On the other hand, the `dbType` annotation is used to test for consistency against the database schema. The following program is rejected at run time because the proposed type annotation does not match the database schema:

```
# let q' = dbSelect db ["name";"ident"]
                       ["people"]
                       [] (int);;

Uncaught exception:
  Failure("Type annotation does not match")
```

The problem occurs because annotation `int` does not match with the actual schema of the query (`string * int`). Thus, any change in the database schema that affects to a query is detected. The exception raised can be caught to perform proper recovery from the error condition:

```
# let q' = try (dbSelect db ["name";"ident"]
                            ["people"]
                            [] (int))
           with DbError -> [] ;;

q' : int list = []
```

In the previous example, the exception is caught and a default value (empty list) is returned if the query cannot be carried out.

4.4 Usage of the primitives

In this section, we introduce some examples using the proposed primitives and the database presented on figure 4. Firstly, we use open a database, binding the delivered database handler to `db`:

```
# let db = dbOpen "sampledb";;
val db : dbconnection = <abstr>
```

Now, we can select the name and the department of all the people who have a qualification greater than 18.5.

```
# let q = dbSelect db ["name"; "dept"]
                      ["people"]
                      [["qualif>=18.5"]; ["dept='cus'"; "dept='acc'"]]
                      (string, string);;

val q : (string * string) list =
     [("Fran Cisco", "acc"); ("Gong Li", "cus"); ("Al Fonso", "cus")]
```

If the query is going to be used more than once, a function can be defined. For instance, the following "procedure" retrieves the names and qualifications of all the people working at the customers or account department with qualification greater than a given argument:

```
# let query q = dbSelect db ["name"; "qualif"]
                            ["people"]
                            [ ["qualif > " ^ string_of_float q];
                              ["dept='cus'"; "dept='acc'"] ]
                      (string, float);;

val query : float -> (string * float) list
```

hence, `query 18.5` will deliver `[("Fran Cisco", 19.5); ("Gong Li", 19.1); ("Al Fonso", 24.6)]`. If we want to change the qualification of a concrete person, we can define:

```
# let change_qualif the_name new_qualif =
        dbUpdate db "people" ["qualif"]
                             [string_of_float qualif]
                             [["name=" ^ the_name]]    (float);;

val change_qualif : string -> float -> unit = <fun>
```

thus we can change the qualification of "Gary Baldi" to 12.4 by using `update_qualif "Gari Baldi" 12.4`.

All the powerful features of functional languages, in particular higher-order functions, can be used to implement complex queries. The following definition uses higher-order functions on lists to upgrade the qualification of all the people in the account or the customers department with qualification greater than a given qualification:

```
# let upgrade_qualif qualif amount =
     let q = query qualif
     in List.iter2 change_qualif
                   (List.map fst q)
                   (List.map ((+.) amount)
                             (List.map snd q));;

val upgrade_qualif : float -> float -> unit = <fun>
```

now we can upgrade by 0.5 the qualification of all the people with a current qualification greater than 18.5 using:

```
# upgrade_qualif 18.5 0.5;;
- : unit = ()

# query 18.5;;
- : (string * float) list =
     [("Fran Cisco", 20.0); ("Gong Li", 19.6); ("Al Fonso", 25.1)]
```

5 Preliminary Results

Even though there are a lot of work in order to improve efficiency of this interface, some measurements have been made in order to compare it with C code with embedded ODBC calls and with MYSQL [1] directly. The example chosen performs a query and then calculates the number of tuples delivered (figure 5). Figure 6 presents the overhead of a naive implementation of the interface with non-optimized definitions. In addition to the overhead introduced by the ODBC interface, it seems that our interface is quite expensive. One of the reasons is due to the inefficient non-tail recursion used to implement the gathering of data from the low-level interface. Figure 7 shows the same benchmark with a simple tail call optimization.

```
open Cdbc;;

let db = dbOpen "sampledb";;

try
  let l = dbSelect db ["ident"; "name"]
                      ["people"]
                      [[]]
                      (int,string)
  in print_int (List.length l)
with DbError m -> print_endline m;
                  flush stdout;;

try dbClose db
with DbError m -> print_endline m;
                  flush stdout;;
```

Figure 5: ODBC-Caml example for measuring

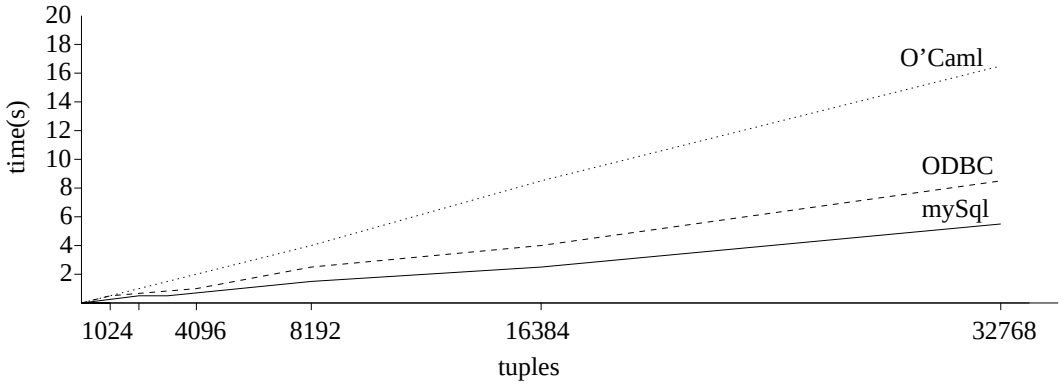

Figure 6: Overhead of a naive implementation of O'Caml-ODBC

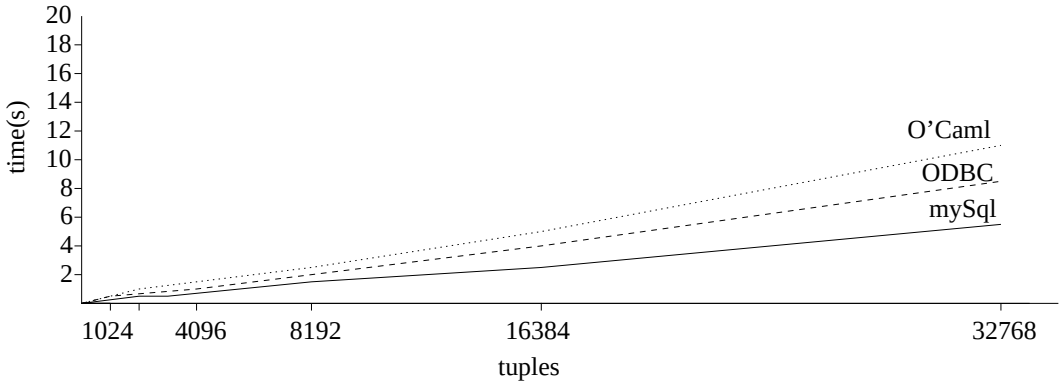

Figure 7: Overheads introduced by ODBC and O'Caml-ODBC

6 Persistence Issues

Persistent data manipulation may be incorporated to a functional language at different levels, ranging from a raw interface with a file system or even a database manager (our approch) to make the language fully persistent. We should notice that our approach is far from *orthogonal persistence* as achieved in the persistent lazy functional language Staple [2]. To point out the way to go, let us discuss some principles exposed in [8]. In persistent programming languages, programs may manipulate data values independently of their persistence and need not refer to the persistence of the values they create (*principle of persistence independence*). Our extension does not provide such capability, being programmer's responsability to store and to retrieve the values of interest. In addition, the only values stored in the database are tuples of O'Caml basic types (`int`, `string`, `float`,...) which are the actual values supported by the database. In order to permit storing any value, including closures and cyclic data structures, (*principle of data type completeness*) some packing of the structures must be performed, similar to the one exposed in [4].

7 Conclusions and Future Work

An extension of a functional language that allows to make queries to a relational manager has been presented. To integrate both packages, an abstraction increasing approach has been used. Our first aim was to get all the power and maturity of a relational manager to server as non volatile storage for functional values, but the abstraction of functional languages may play an interesting role to reduce the effort required to develop data-access procedures. Moreover, it can help to build *recursive* queries, an interesting future line of research.

Future guidelines must include the development of a real application, that will serve as a real-world benchmark for this extension. In concrete, we are interested in knowing the behaviour of the library with queries that involve dealing with big amounts of data. The evaluation policy of O'Caml forces that results of the queries must be converted completely to lists, which is not appropriate in some settings. A possible solution would be the use of *lazy lists* constructed on demand. The garbage collector will reclaim the list nodes that are no longer needed. In order to use this interface easily, a web interface is being developed, using a CGI interface. We expect to develop real-world O'Caml

applications using this www interface and the proposed interface with ODBC.

As a final research interest, we would like to include persistent capabilities in the language and combine them with database storage, in the line commented in section 6, as well as distributed processing.

References

[1] D. Axmark, M. Widenius, and K. Aldale. *MySQL Reference Manual for version 3.21.19-beta.* T.c.X. DataKonsulAB.

[2] A. J. T. Davie and D. J. McNally. Statically typed applicative persistent language environment (STAPLE) reference manual. Research Report CS/90/14, Department of Mathematical and Computational Sciences, University of St. Andrews, St. Andrews, 1990.

[3] J.L. Freire, V.M. Gulías, and X.M. López. On the Functional Approach to RDBMS. In *Data Management Systems, Proc. of the 3rd Int. Workshop on Information Technology, BIWIT'97, July 2–4, 1997, Biarritz, France*, pages 169–177, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[4] K. Hammond, P. Trinder, P. Sansom, and D. McNally. *Improving Persistent Data Manipulation for Functional Languages.* Springer-Verlag, New York, NY, 1992. Springer-Verlag Workshops in Computing.

[5] B. Jepson. *The FreeODBC Pages.* `http://users.ids.net/ bjepson/freeODBC/`.

[6] T. Johnston and M. Osborne. *ODBC Developers Guide.* Howard W. Sams & Company, 1994.

[7] X. Leroy. *The Objective Caml System, release 1.05.* INRIA, 1997.

[8] A. J. T. McNally and Davie D. J. Two models for integrating persistence and lazy functional languages. *SIGPLAN Notices*, 26(5):43–52, [5] 1991.

[9] D. J. McNally, S. Joosten, and A. J. T. Davie. Persistent functional programming. In *Fourth Int'l Workshop on Persistent Object Sys.*, page 59, Martha's Vineyard, MA, September 1990.

[10] R.S. Nickhil. *Functional Databases, Functional Languages.* Atkinson, M.P., Buneman, P. & Morrison, R., Springer, 1988.

[11] A. Poulovassilis and P. King. Extending the functional data model to computational completeness. *Lecture Notes in CS*, 416:75, March 1990.

[12] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6:140–173, 1981.

[13] M. Stegman, R. Signore, and J. Creamer. *The ODBC Solution, Open Database Connectivity in Distributed Environments.* McGraw-Hill, 1995.

[14] A. Yu and J. Chen. The Postgres95 user manual. Dept. of EECS, University of California at Berkeley, 1995.