A Brief Introduction to PG’OCAML*

Dario Teixeira
(dario.teixeira@yahoo.com)

Version 0.94
5th April 2010

Contents

1. Introduction

PG’OCAML, by Richard W. M. Jones (rich@annexia.org), provides an interface to POSTGRESQL databases for OCAML applications [1, 2, 3]. It uses CAMLP4 to extend the OCAML syntax, enabling one to directly embed SQL statements inside the OCAML code [4]. Moreover, it uses the *describe* feature of POSTGRESQL to obtain type information about the database. This allows PG’OCAML to check **at compile-time** if the programme is indeed consistent with the database structure. This type-safe database access is the primary advantage that PG’OCAML has over other POSTGRESQL bindings for OCAML.

Unfortunately, PG’OCAML is rather lacking on the documentation front. This document aims to fill that gap, by providing an overview of the capabilities of the library, usage examples, and solutions to potential pitfalls. Moreover, it also addresses the installation of PG’OCAML, how to compile programmes that make use of the library, and the correspondence between POSTGRESQL data types and their OCAML counterparts.

The latest version of this document can always be found at the following address:
`http://dario.dse.nl/projects/pgoctut/`

2. Installation

You are strongly advised to use OCAMLFIND to aid in the management of OCAML packages. The instructions on this document will therefore assume that you are using OCAMLFIND, and that you will have PG'OCAML installed in a manner consistent with it. Fortunately, the makefile included with the source code of PG'OCAML already has provisions for adding PG'OCAML to OCAMLFIND's repository. After you have built the PG'OCAML library (typically with `make all`), simply run `make findlib_install` to perform the installation. This should create a `pgocaml` directory under the appropriate branch of OCAMLFIND's repository (normally under a directory named `site-lib` if you are using GODI). In this directory you will find the compiled PG'OCAML libraries, plus the `META` file with special instructions for OCAMLFIND.

3. Compilation of projects using PG'OCAML

Figure 1 lists a basic makefile for compiling a project *test* that makes use of PG'OCAML. Note that the CAMLP4 syntax extension used by the PG'OCAML can be handled in a fairly straightforward manner thanks to OCAMLFIND. Note also that the sub-package `pgocaml.syntax` must be used during the compilation stage of code that makes use of the CAMLP4 syntax extensions. This sub-package is of course unnecessary during the linking stage.

```
PROJECT  := test
LINK_PKG := pgocaml
COMP_PKG := pgocaml,pgocaml.syntax

all: $(PROJECT)

$(PROJECT): $(PROJECT).cmo
	ocamlfind ocamlc -package $(LINK_PKG) -linkpkg -o $@ $<

$(PROJECT).cmo: $(PROJECT).ml
	ocamlfind ocamlc -package $(COMP_PKG) -syntax camlp4o -c $<
```

Figure 1: A simple Makefile to compile PG'OCAML projects. Note that OCAMLFIND must be installed in the system. In addition, the CAMLP4 preprocessor is invoked during the compilation stage.

4. Basic usage

Figure 2 lists a very simple programme that uses PG'OCAML. In this section we shall dissect this programme function by function, thereby introducing the basic principles behind PG'OCAML. Note that in order for the programme to compile and run, the POSTGRESQL *Postmaster*[1] must be running on the local host, and there must be a database with the same name as your user's defined within the system (run `createdb `whoami`` if that is not the case). The reasons behind this should be made clear before this section is through.

From a bird's eye perspective, what stands out immediately is the embedding of SQL statements inside the OCAML code. PG'OCAML can deal with pretty much all valid SQL statements, including sub-selects. Though not quite as conspicuous, a more careful look at the code will show that PG'OCAML must somehow be extending the type-safety of OCAML to the embedded statements. Note that the `users` table is declared to have three columns, respectively of SQL types `serial`, `text`, and `int` (all of them `not null`). If one were to run `ocamlc -i` on this code, the signature of the `print_user` function would equal `val print_user : int32 * string * int32 -> unit`, indicating that the

[1]The *Postmaster* is the frontend process that manages connections to the POSTGRESQL databases.

system was able to infer the correct OCAML types that correspond to the POSTGRESQL types declared in the embedded statements (see Section 5 for a more thorough description of the correspondence between POSTGRESQL and OCAML data types).

```
let create_table dbh =
      PGSQL(dbh) "execute" "create temporary table users
      (
      id          serial not null primary key,
      name        text not null,
      age         int not null
      )"

let insert_user dbh name age =
      PGSQL(dbh) "INSERT INTO users (name, age)
                  VALUES ($name, $age)"

let get_users dbh =
      PGSQL(dbh) "SELECT id, name, age FROM users"

let print_user (id, name, age) =
      Printf.printf "Id: %ld  Name: %s  Age: %ld \n" id name age

let _ =
      let dbh = PGOCaml.connect () in

      let () = create_table dbh in

      let () =
            insert_user dbh "John" 30l;
            insert_user dbh "Mary" 40l;
            insert_user dbh "Mark" 42l in

      List.iter print_user (get_users dbh)
```

Figure 2: A simple programme using PG'OCAML. Note the syntax extension enabling the embedding of SQL statements inside OCAML code.

As for the syntax extension, it takes the form of the macro `PGSQL`, followed by the database handle between parentheses, an optional sequence of strings with the statement flags, and a final, mandatory string with the actual SQL statement. You can see the extension in use in functions `create_table`, `insert_user`, and `get_users`.

4.1. Statement flags and environment variables

The sage reader will have come to the conclusion that in order for the compiler to verify the correct match between the database structure and the types used in the programme, PG'OCAML must have access to the database **at compile-time**. That is indeed true. Moreover, it follows that there must be at least one mechanism that allows the programmer to inform PG'OCAML where the relevant POSTGRESQL *Postmaster* is located, and how the target database should be accessed. In fact, PG'OCAML provides not one, but two different and alternative mechanisms for this purpose: environment variables, and statement flags.

Environment variables are set via the normal mechanism available in the operating system. Due to their global nature, they apply to **all** PG'OCAML statements in the programme. Moreover, they can be used both at compile-time and runtime. As for statement flags, they take the form of string constants placed before the SQL statement proper. They are therefore valid only for that statement. In the example shown in Figure 2, only one statement flag is used: the `"execute"` placed before the SQL statement in function `create_table`.

Table 1 lists all statement flags and associated environment variables. A statement flag will override the corresponding environment variable, and lacking both, the built-in defaults are used. You can now understand why the example in Figure 2 requires that a database with your user name exists in the local host: since we have not declared neither host, nor user, nor database, the default is to use the local machine, your user name, and a database named after the user, respectively.

Statement flag (Environment variable)	**Observations**
`host=...` (PGHOST)	If the host is not specified, the connection will default to the localhost, using a UNIX domain socket for communication.
`port=...` (PGPORT)	If the port number is not specified, the default is 5432. Note that the port number is only used if the host is specified.
`user=...` (PGUSER)	If no user name is specified, the default is to use the current UNIX user name. If the latter is also unavailable, `postgres` is tried.
`password=...` (PGPASSWORD)	The password used to authenticate the user, if the POSTGRESQL configuration so requires.
`database=...` (PGDATABASE)	The name of the database we wish to connect to. If not specified, a database with the same name as the user is tried.
`unix_domain_socket_dir=...` (UNIX_DOMAIN_SOCKET_DIR)	The directory where the UNIX domain socket can be located. In a DEBIAN system, for instance, this directory is typically `/var/run/postgresql/`.
`execute` (N/A)	Tells PG'OCAML that the statement should be executed immediately (at compile-time). This flag only makes sense on a statement by statement basis, and therefore has no equivalent environment variable.
`nullable-results` (N/A)	Disables the *nullability* heuristics for all columns. For details consult the `BUGS.txt` file included with PG'OCAML.

Table 1: Statement flags and environment variables. Note that statement flags are only valid at compile-time and on a statement by statement basis. Environment variables, on the other hand, are valid both at compile-time and runtime; moreover, they apply globally, to all statements.

4.2. The connection handle

At runtime, before any SQL statements can be issued, you must create a connection handler to the POSTGRESQL database. This handler is created by the `PGOCaml.connect` function, whose signature is shown in Figure 3. Note that the optional parameters for this function mirror those available via the environment variables. In the code shown in Figure 2, the connection handle `dbh` is created by the first statement of the top-level anonymous let-binding.

At this point, the reader may be wondering if there is not redundancy between the parameters of

```
val connect :
        ?host:string ->
        ?port:int ->
        ?user:string ->
        ?password:string ->
        ?database:string ->
        ?unix_domain_socket_dir:string ->
        unit -> 'a t
```

Figure 3: The signature of function `PGOCaml.connect`, used to create a database connection handle.

the `connect` function and the already discussed statement flags and environment variables. Partly yes, though there are still good reasons why `connect` accepts these parameters as well. First, bear in mind that the statement flags are valid only at compile-time, while the parameters to `connect` are used only at runtime. Second, though environment variables can be used both at compile and runtime, they require an action by the user to set them up. By passing the connection parameters directly in the `connect` function, the programme is able to run correctly even if the user forgets to set the environment variables. Moreover, the parameters to `connect` trump environment variable definitions.

4.3. Parameters and return values of SQL statements

As shown in function `insert_user`, the basic notation for passing an OCAML value to an SQL statement is to simply prefix the name of the value with the dollar sign `$` (optional and array types require a different notation, discussed in Section 5.2.

As for the return type of the embedded SQL statements, they match fairly closely the natural types one would expect. Statements that return no data (such as the `INSERT` statement in function `insert_user`) have type `unit`. Likewise, `SELECT` statements (such as the one in function `get_users`) will typically return a list of tuples.

If in doubt about the actual return type of a more complex statement (such as one involving SQL aggregate functions), then `ocamlc -i` is your friend. Consider, for example, that we were to add to the programme the function `get_aggregates` listed in Figure 4. It is far from obvious what the actual signature of this function is. Thankfully, the figure shows also the signature produced by `ocamlc -i`, telling us that the function returns a list (typically composed of a single element) of tuples. The tuples are formed by two optional types: an `int64` corresponding to the number of rows in the table [2], and a float corresponding to the average of the user ages.

```
let get_aggregates dbh =
    PGSQL(dbh) "SELECT COUNT (id), AVG (age) FROM users"

val get_aggregates :
    (string, bool) Hashtbl.t PGOCaml.t -> (int64 option * float option) list
```

Figure 4: A new function `get_aggregates` that returns the number of rows in the `users` table and the average of the `age` column. Note that the signature of this function is far from obvious, so `ocamlc -i` can be of help.

[2] In fact, there is no limit to the number of rows that a POSTGRESQL database can hold. It just so happens that `int64` is the largest integer type that OCAML can handle.

5. Data types

The translation between POSTGRESQL and OCAML types is not as straightforward as one might think. Consider for example that due to requirements of the garbage collector, the `int` type in OCAML is actually 31 bits long, instead of the 32 bits integers commonly found in other languages and in POSTGRESQL's own `int` type.

PG'OCAML chooses safety and correctness over potential performance gains. Therefore, POSTGRESQL's `int` type is mapped into OCAML's `int32`. Table 2 lists the correspondence between all the POSTGRESQL types currently supported by PG'OCAML and their OCAML counterparts. Note in particular that all character types are mapped onto OCAML's `string`, and that thanks to the facilities offered by the CALENDAR library [5], it is also possible to do a type-safe and semantically correct mapping of the time and date types.

5.1. Handling optional types

SQL features the possibility of declaring certain columns as `NULL` (this is in fact the default if the column is not explicitly declared `NOT NULL`). These `NULL` values in SQL represent essentially the same concept as the `None` in OCAML's optional types. Therefore, it should not come as a surprise that PG'OCAML uses optional types to represent SQL columns that accept `NULL` values.

```
let create_table dbh =
      PGSQL(dbh) "execute" "create temporary table users
      (
      id           serial not null primary key,
      name         text not null,
      age          int
      )"

let insert_user dbh name age =
      PGSQL(dbh) "INSERT INTO users (name, age)
                  VALUES ($name, $?age)"

let get_users dbh =
      PGSQL(dbh) "SELECT id, name, age FROM users"

let print_user (id, name, age) =
      let age_str = match age with
            | Some number     -> Int32.to_string number
            | None            -> "(no age)"
      in
      Printf.printf "Id: %ld  Name: %s  Age: %s \n" id name age_str

let _ =
      let dbh = PGOCaml.connect () in

      let () = create_table dbh in

      let () =
            insert_user dbh "John" (Some 30_l);
            insert_user dbh "Mary" (Some 40_l);
            insert_user dbh "Mark" None in

      List.iter print_user (get_users dbh)
```

Figure 5: An extended example, making use of optional types. Note that because the POSTGRESQL type of column `age` now accepts `NULL` values, its corresponding OCAML type has been changed to `int32 option`.

POSTGRESQL	OCAML
Numeric types	
int2, smallint	PGOCaml.int16 [a]
int4, int, integer	int32
serial	int32
int8, bigint	int64
decimal, numeric	float
float8, float, double precision	float
float4, real	float
Character types	
char, character	string
varchar, character varying	string
text	string
Time and date types	
date	Date.t
interval	Calendar.Period.t
time	Time.t
timestamp	Calendar.t
timestamptz	PGOCaml.timestamptz [b]
Blob types	
bytea	PGOCaml.bytea [c]
Logical types	
bool, boolean	bool
Array types	
int[]	PGOCaml.int32_array [d]

Table 2: Correspondence between POSTGRESQL types and their OCAML counterparts. Note that most integer types are mapped onto either `int32` or `int64`, to avoid overflowing the 31 bits of OCAML's native `int` type. As for character types, they are all mapped onto OCAML `string`. At last, note that temporal types are mapped onto the facilities offered by the CALENDAR library.

[a] `PGOCaml.int16` is defined as `int`.
[b] `PGOCaml.timestamptz` is defined as `Calendar.t * Time_Zone.t`.
[c] `PGOCaml.bytea` is defined as `string`.
[d] `PGOCaml.int32_array` is defined as `int32 array`.

Figure 5 lists a modified version of our original programme. Note that we have made `NULL` values acceptable for the column `age`. As a consequence, the associated OCAML type is now `int32 option`. You will notice that function `print_user` has some extra code to handle for the possibility of no age being defined. Note also that when referencing an optional type inside an embedded statement, the notation `$?` should be used instead of the plain `$`.

5.2. Array types and list expressions

Figure 6 lists further modifications to our original programme. Besides the changes incorporated in Figure 5, the reader will notice that we added a new column to the table, of type `int[]`. POSTGRESQL supports arrays are column types, and PG'OCAML also has limited support for them. Note also that we added two new functions, `get_2_users` and `get_n_users`, both using list expressions.

```
let create_table dbh =
      PGSQL(dbh) "execute" "create temporary table users
      (
      id          serial not null primary key,
      name        text not null,
      age         int,
      votes       int[]
      )"

let insert_user dbh name age votes =
      PGSQL(dbh) "INSERT INTO users (name, age, votes)
                  VALUES ($name, $?age, $votes)"

let get_users dbh =
      PGSQL(dbh) "SELECT id, name, age FROM users"

let get_2_users dbh =
      PGSQL(dbh) "SELECT id, name, age FROM users WHERE id IN (1, 2)"

let get_n_users dbh user_ids =
      PGSQL(dbh) "SELECT id, name, age FROM users WHERE id IN $@user_ids"

let print_user (id, name, age) =
      let age_str = match age with
            | Some number     -> Int32.to_string number
            | None            -> "(no age)"
      in
      Printf.printf "Id: %ld  Name: %s  Age: %s \n"
            id name age_str

let _ =
      let dbh = PGOCaml.connect () in

      let () = create_table dbh in

      let () =
            insert_user dbh "John" (Some 30_l) [| 10_l; 15_l |];
            insert_user dbh "Mary" (Some 40_l) [| 16_l |];
            insert_user dbh "Mark" None [| |] in

      List.iter print_user (get_users dbh);
      List.iter print_user (get_2_users dbh);
      List.iter print_user (get_n_users dbh [2_l; 3_l])
```

Figure 6: An example using array types and list expressions. While the former are referred to just like any other POSTGRESQL type, the latter require the use of the special `$@` notation if used programatically, as illustrated by function `get_n_users` (note that this function will cause a runtime exception if the list `user_ids` happens to be empty; a workaround is shown in Figure 7).

It is important that array types and list expressions are not confused. The former are used in PG'OCAML just like any other type; note that we use the basic `$` notation to refer to column `votes`. As for list expressions (the `(1, 2)` used in function `get_2_users`, for example) they require a special notation if they are created programatically. Function `get_n_users` illustrates this aspect: note the use of the $@ notation.

While certainly useful, the programatic use of list expressions has a number of caveats that the user should be aware of. These stem from shortcomings in the SQL standard, bugs in older versions (pre 8.x) of POSTGRESQL, and limitations inherent to the way PG'OCAML prepares SQL statements. The user is strongly advised to take heed of these warnings:

a) Due to an unfortunate lack of foresight, the SQL standard does not accept empty list expressions. Therefore, if we were to replace the `(1, 2)` list in function `get_2_users` with the empty list `()`, compilation would fail with a syntax error. More worryingly, the programatic use of list expressions (as exemplified by function `get_n_users`) brings forth the very real danger of an empty list being passed to an SQL statement, causing a syntax error complaint from the database server and consequent exception at runtime. You are therefore strongly advised to guard against this possibility by checking beforehand if the list is empty. A revised, correct version of function `get_n_users` is shown in Figure 7.

```
let get_n_users dbh user_ids =
    match user_ids with
    | []    -> []
    | _     -> PGSQL(dbh) "SELECT id, name, age FROM users WHERE id IN $@user_ids"
```

Figure 7: A revised version of the function `get_n_users`. Unfortunately, the SQL standard does not accept empty list expressions. Therefore, when using the `$@` notation to programatically insert a list expression into a statement, the user is strongly advised to check against the empty case to avoid a runtime exception.

b) Particularly in older versions of POSTGRESQL (before the 8.x series), large list expressions could cause serious performance and/or crashes in the database server [6]. You are therefore advised to upgrade to newer versions of POSTGRESQL or to be careful with the size of the list expressions used programatically.

c) Due to the way POSTGRESQL prepared statements work, PG'OCAML is forced to make a prepared statement for each length of a programatic list expression used. Therefore, if we were to invoke function `get_n_users` successively with lists `[10_l]`, `[10_l; 11_l]`, and `[10_l; 11_l; 12_l]`, PG'OCAML would have to prepare and store each of the following statements:

```
SELECT id, name, age FROM users WHERE id IN ($1)
SELECT id, name, age FROM users WHERE id IN ($1, $2)
SELECT id, name, age FROM users WHERE id IN ($1, $2, $3)
```

The astute observer will have noticed that if the size of the list is potentially very large, and if successive invocations of the function happen for varying sizes of the list, then the amount of memory spent on the prepared statements can easily grow out of hand. There is no easy workaround this issue, so the user should keep this problem in mind.

6. Frequently Asked Questions

6.1. Are there provisions against SQL injections?

Yes. Internally, PG'OCAML uses so-called *prepared statements* to operate on the database. What this means is that a statement is first prepared with placeholders instead of the actual parameters. The database then parses and creates a plan for the statement. It is only after this that the actual parameters are fed to the database. Not only does this procedure prevent the user to inject SQL statements, but it also saves the database engine the effort of parsing and planning the same statement each time it is issued.

6.2. Can I dynamically construct SQL statements?

No. Bear in mind that PG'OCAML must have access to the statement at compile-time. Therefore, you cannot build a statement dynamically from smaller pieces.

6.3. Can `select` statements return a list of records instead of tuples?

This is not possible at the moment. If you need to convert the list of tuples returned by a PG'OCAML statement, you need to run `List.map` on the returned list, and use a constructor function to convert a tuple into a record.

6.4. Is PG'OCAML thread-safe?

Yes, with some reservations. Internally, each database connection handle (the type returned by the function `PGOCaml.connect`) is a hash table produced by the module `Hashtbl`. This hash table contains the MD5 hashes of the SQL prepared statements, which are used to uniquely identify each prepared statement with the database server. Now, if two threads are simultaneously executing the same statement, and they both discover it is not in the hash table, then they will both compute its MD5 hash and use it to store the prepared statement in the database. The problem is that POSTGRESQL cannot accept two prepared statements with the same identifier for the same connection, and will complain. Therefore, if you intend to use PG'OCAML in a threaded programme, make sure that each thread uses a separate connection handler.

Acknowledgements

I would like to thank Richard W. M. Jones (rich@annexia.org), the author of PG'OCAML, for reviewing the early drafts of this document and for answering all my doubts concerning the library.

References

[1] http://merjis.com/developers/pgocaml

[2] http://www.postgresql.org/

[3] http://caml.inria.fr/

[4] http://caml.inria.fr/pub/old_caml_site/camlp4/index.html

[5] http://www.lri.fr/~signoles/prog.en.html

[6] http://svr5.postgresql.org/pgsql-sql/2007-02/msg00251.php

Revision history

Version 0.92 (2007-09-06) Added a list of caveats to the programatic use of list expressions (thank you to Richard W. M. Jones (rich@annexia.org) for pointing this out).

Version 0.91 (2007-09-05) First public release.