Type System for SQL, and SQL for OCaml

Zachary Bechhoefer, Noah Mintz Roberts

May 4, 2018

The purpose of this paper is to explore implementation type systems for SQL semantics, and their implementation in OCaml. This investigation arises from research on implementing a database interface for OCaml, necessitating a type system. To date, there are numerous proposals for SQL type systems, despite the rigid conventions of SQL syntax. Conversely, we would expect a âbest type systemâ to have overshadowed others. The presence of rigorously defined SQL type systems, contrasted by the relatively few number of OCaml interfaces begs the question of what technicalities are impeding this utility extension and others, to the language. Difficulty with establishing firm conventions for database access is not confined to OCaml. Haskell answers the question of database interfacing with Yesod, a web framework with a database interface demanding ungodly amounts of configuration, and for a very strongly typed language, a larger code and software vocabulary than is practical for something conceptually simple: database interfacing. Yesodâs Haskell alternative is HDBC, which simply processes queries from strings. Even languages like Kotlin resorts to using strings to make database queries. Whence the lack of functional database interfaces for OCaml?

There is one existing solution to database interfacing with OCaml, Macaque by Ocsigen. Macaque is a fine, if not heavyweight OCaml database access solution (1). It is just one solution, and too heavyweight to be appropriate for all applications, and it is not simple. This should not be a problem, because in most programming languages used commercially. In Python for example, the database connection or any query to any SQL database is just one line each. Programming languages should be reflexively adaptable to a diverse set of use cases, and database access is not an exception to the rule.

Our investigation into extending OCaml with lightweight database access begins with OCaml C. While OCaml, at this stage of maturity, does have a hardware-level type system, it would not be practical for the average developer to learn. Instead the OCaml C library, caml/mlvalues.h and others give developers low level access to the computer from OCaml. The caml/mlvalues.h library has a few key features used in our database implementation:

- The value struct, a struct that holds an arbitrary number of parameters, and is a fill-in for all OCaml types.
- Field(value v, unsigned long n): The method for dereferencing fields. Tuples place each of their values in one parameter, linked list elements reference their values at index 0, and a pointer to the following element at index 1.
- "Val_[unit | long | <type> | ...]:" Fields store values as raw untyped data, which must be referencing using special value-type referencing functions.

In addition, because of OCamlâs garbage cleanup and side-effect-lessness, OCaml C enforces special rules for allocating non-pointer data (by throwing a Segmentation Fault (Core dumped) error without any explanation even with -Wall). Namely, when operating on allocated memory in OCaml C, the user must use CAMLlocal(value) to specify allocated data, CAMLparam to create parameters used in function calls, and CAMLreturn to pass allocated data into the OCaml program. The .ml file then calls compiled C functions like:

```
external sql_connect: string -> string -> string -> dbconn = "sql_connect"
```

While C for OCaml is very loosely typed, with all value structs being interchangeable, the type checking is enforced here. This means that writing type-safe libraries extended with OCaml C requires refactoring as much code into the ML as possible. Notably, pointers to structs can be held in memory in abstract types such as dbconn above, declared as just âtype dbconn.â

The final challenge to extending OCaml with C is linking and compiling the database access library. C programs accessing databases must often do so using a special linker option for that database software. Even among SQL querying languages, each database is accessed using its own library and linker option, requiring case-by-case C extensions for each database software. In our research, we wrote some code for an access library to MySQL, which was compiled using:

```
ocamlopt main.ml -cc "gcc sql_conn.c -lmysqlclient"
```

Which is the OCaml C compiler nesting its C such that the SQL access linker option is applied to main.ml. This is necessary because SQL linker options, while only adding functionality to C, must be available to the executable file. There are multiple low level OCaml compilation command, including ocamlrun and ocamlc which execute and create bytecode respectively. Each of these commands has many compiler options, and Makefiles for C extensions to OCaml have been observed to be very large, integrating functionality from many other C and .ML libraries, requiring their own compilation rules.

In terms of implementation, our research has clarified challenges to the availability of database access libraries for OCaml. While OCaml has an excellent, extensive type system for system calls, and a more accessible C library, the challenge appears to come from the varied SQL access rules. Implementing a pan-SQL access library from any low level libraries requires a lot of work, and many dedicated developers by extension. Those developers in turn are motivated by the industry popularity of the software. This explains the database support for Python, Ruby, and .NET, and also indicates that a well liked language comes before the development support, instead of vice versa. This could be bad news for application development in OCaml, which may never get in the loop of being liked and having support. The excellent system calls type system and type checking for OCaml makes in excellent for implementing new languages, but it may only live on through those languages it compiles. This is very believable, since Prolog followed the same path, ultimately dying out, but being use to write the high-utility Erlang language (with compiler is now written in OCaml).

While fully implementing a type system for OCaml SQL statements worked out to be infeasible for the scope of this project, we did look at our options. We settled on the following type system to model our interface after.

Relational Database's Beta ranges over entries:

$\tau : \beta := T_1 as N_1, ..., T_k as N_k \ for \ k > 0, \tau = (\tau_1 to \tau_k), \beta = (N_1 to N_k)$

Relational Database's Alpha ranges over terms. Each Entry has k terms:

$\alpha : \beta' := t_1 as N_1{}', ..., t_m as N_k{}' \ for: \ m > 0, \alpha = (\tau_1 to \tau_m), \beta' = (N_1{}' to N_k{}')$

n*m*k values associated this way

Of course none of these values can be zero for a proper database to form.

One of the problems with SQL Standard is that the semantics of queries is non com-positional, semantically, a query can behave differently depending on the context in which it occurs.

Conditions for our SQL Query language are important to outline:

Our conditions all boil down to boolean states, a conditional is either True,False,or a predicate that evaluates to a boolean True,False.

$\theta := True | False | P(t_1 to t_k), \ Where \ P \ is \ a \ collection \ of \ predicates$

The predicate states are typical in any boolean system. Membership vs. negated membership,

$| \ t \ is \ [NOT] \ NULL$

A querry evaluation where (assuming t was a member) t-bar represents the conglomeration of values from t. So our value either, is/is not in Q, this assumes Q exists,

$| \ t - bar \ is \ [NOT] \ in \ Q \ | \ Exists \ Q$

Theta has union, intersection, and negation as part of it's conditional functionality,

$| \ \theta[AND][OR]\theta$

$| \ [NOT]\theta$

For SQL Queries, it is different from what we have seen before:

Q:= SELECT[DISTINCT]$\alpha : \beta' \ from \ \tau : \beta \ where \ \Theta$

Given our conditional theta, we search the entries terms (Alpha:Beta') from the full database entries (Tau:Beta). This can either be distinct or non-distinct selection.

Another usage,

SELECT[DISTINCT]*from$(\tau : \beta) where \ \Theta$

In this usage, '*' simply implies the return of all the entries terms from the full database, so if you grabbed a dog â You'd get Color, Name, Size or whatever your terms happened to be.

As mentioned before with the conditionals, SQL queries have specific clauses for different combinational structures:

Q(UNION | $INTERCEPT$ | $EXCEPT)[ALL]Q$

The SQL Union operator returns rows from both databases. If used by itself, UNION returns a distinct list of rows. Using

UNION ALL, returns all rows from both databases

The SQL Intercept operator returns rows that the two databases share in common, as well as the ones that are unique in between them.

The SQL Except operator returns rows from the first databases that aren't also in the second database.

Small Steps:

If Q steps to some E, then Q exists, and the E it steps to is non-empty

If Q$\xrightarrow{n} E\ |EXISTS\ Q \xrightarrow{n}\ NOT\ EMPTY(E)$

It is possible if our base value, say b is already boolean, T or F, then b steps to b in any number of steps

b$\xrightarrow{n} b|b = TRUE\ or\ FALSE$

If t IS NULL, it steps to null t-hat if t stepped to t hat before:

t IS NULL $\xrightarrow{n} null(\hat{t})|IF\ t \xrightarrow{n} \hat{t}$

If t IS NOT NULL, it steps to null t-hat if t stepped to t hat before:

t IS NOT NULL $\xrightarrow{n} NOT\ null(\hat{t})|IF\ t \xrightarrow{n} \hat{t}$

We can step the whole predicate base to t-hat if entry I steps to I-hat and entry I' steps to I'-hat:

P(t1,...,ti)$\xrightarrow{n} P(\hat{t1}, ..., \hat{ti})|IF\ ti \xrightarrow{n} \hat{ti}, ti' \xrightarrow{n} \hat{ti}'$

While in the state with conditionals (theta) evaluating to theta' we have:

$|\ For\ \Theta_1 \xrightarrow{n} \Theta_1', \Theta_2 \xrightarrow{n} \Theta_2'|$

$|\ \Theta_1\ AND\ \Theta_2 \xrightarrow{n} \Theta_1' \wedge \Theta_2'|$

$|\ \Theta_1\ OR\ \Theta_2 \xrightarrow{n} \Theta_1' \vee \Theta_2'|$

$|\ NOT\ \Theta_1 \xrightarrow{n} NOT\ \Theta_1'|$

Such that the union of theta1 and theta2 can evaluate to the union of theta1' and theta2' over n steps. The same applies for intersection and NOT, but not EXCEPT

References

[1] Tutorial Your First Application.Ocsigen, ocsigen.org/tuto/4.0/manual/macaque.

[2] Python Database Objects (DBO). Python Database Objects, python-dbo.sourceforge.net/.