Statically-typed value persistence for ML

Thomas Gazagnaire

Citrix Systems R&D
Building 101, Cambridge Science Park, Milton Road,
Cambridge CB4 0FY, UK
thomas.gazagnaire@citrix.com

Anil Madhavapeddy

Computer Laboratory, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK
avsm2@cl.cam.ac.uk

Abstract

We present a set of extensions to the OCaml language which add support for statically generating typed accessor functions for persisting and communicating a large subset of OCaml types. The extensions do not require any compiler modifications and are implemented using the `camlp4` AST transformer. We describe our utility libraries which permit analyzing OCaml values and type declarations from library code without the complexity of a fully-staged system such as MetaOCaml.

Keywords: ocaml, metaprogramming, generative, database, network, sql

1 Introduction

ML compilers can generate very efficient code due to their use of static typing and simple run-time requirements, making them ideal for constructing reliable, high-performance servers [6]. However, conversions between the variety of values that these servers manipulate is a largely manual process that can be tedious and error-prone. It can be automated via *staged programming* to generate and execute code fragments as part of the compilation process [10], but introducing N-stage languages such as MetaOCaml introduces additional complexity in the tool-chain.

We describe a simpler 2-stage transformer that is sufficient for generating efficient storage code for ML programs. Our implementation uses OCaml [5] and `camlp4` to: (*i*) parse a large subset of OCaml types and values to a more succinct form than the syntax tree which `camlp4` provides; (*ii*) implement an Object-Relational Mapping (ORM) which defines an efficient conversion to and from SQL; (*iii*) provide a syntax extension to augment type definitions in existing code.

End-user programmers use the same types and values as they did previously, but additional functions are generated to persist and retrieve these values from the

database. Parts of the extension have been developed for use in the Xen [2] control toolstack—a large, complex OCaml application that has been developed since 2006. The Xen toolstack runs in an embedded Linux distribution and controls virtual machine activity across pools of physical machines, and so requires efficient storage and I/O. One of the benefits of our approach is that it works as a library to the standard OCaml distribution.

In the remainder of the paper, we first describe the type parsing (§2) and value introspection libraries (§3). Then we show its use in the ORM layer (§4), and finally an example of a simple photo gallery (§5).

2 Type Introspection

In this section we describe *type-of*, a library which make type introspection available in OCaml. It uses the `type-conv` framework [8] to convert an ML type definition annotated with the keyword `type_of` into a finite ML value representing that type and usable at runtime. For a given type `t`, the library will generate a value named `type_of_t` of type `Type.t` defined as follows:

```
module Type = struct
    type t =
        | Unit | Bool | Float | Char | String
        | Int of int option
        | Arrow of t × t
        | Option of t
        | Enum of t
        | Tuple of t list
        | Dict of ( string × [‘RW|‘RO] × t ) list
        | Sum of ( string × t list ) list
        | Ext of string × t
        | Rec of string × t
        | Var of string
end
```

OCaml

2.1 Basic Types

The basic types are similar to the usual OCaml basic types, i.e. `Bool`, `Float`, `Char` and `String` which can be composed using `Arrow`. Integers have an addition bit-width range parameter that can be 31-, 32-, 63- or 64-bit depending on the exact OCaml type and host architecture, or unlimited for `BigInt` types. These basic types can be composed to form either a tuple with `Tuple`, a record or an object with `Dict`, or a (polymorphic) variant with `Sum`.

For example, consider the basic type definitions:

```
type tuple = int32 × string with type_of
type record = { mutable foo : string } with type_of
type variant = Foo | Bar of bool with type_of
```

OCaml

These definitions will generate the following ML code:

```
let type_of_tuple = Ext ( “tuple”, Tuple [ Int (Some 32); String ] )
let type_of_record = Ext ( “record”, Dict [ (“foo”, ‘RW, String) ] )
let type_of_variant = Ext ( “variant”, Sum [ (“Foo”, []) ; (“Bar”, [Bool]) ] )
```

OCaml

2.2 Type Variables

Types variables are handled quite naturally, by induction on the type structure in which they are used. Hence, a recursive type definition as `type t = x option with type_of` will generate the following ML expression : `let type_of_t = Ext ( "t", Option type_of_x)`. In this case, `type_of_x` has to be defined for the program to compile. This definition may have either been automatically generated previously using the *type-of* library, or have been defined by the user. The later option makes the *type-of* library easily extensible, especially for abstract types.

2.3 Recursive Types

Recursive types are handled carefully in order to always keep a finite representation of the ML type. This is done using the constructors `Rec(v,t)` and `Var v`. `Rec(v, t)` can be understood as the binding of the type variable `v` to the type expression `t`; `Var v` always appears in the scope of a corresponding `Rec(v,t)` and is equivalent to the substitution of `Var v` by `t` in `t`.

The following example show the code which is automatically generated for simple recursive types:

```
(* User-defined datatype *)                                          OCaml
type t = { x : x } and x = { t : t } with type_of

(* Auto-generated code *)
let type_of_t = Rec ( "t", Dict [ "x", Ext ( "x", Dict [ "t", Var "t"] ) ] )
let type_of_x = Rec ( "x", Dict [ "t", Ext ( "t", Dict [ "x", Var "x"] ) ] )
```

3 Value Introspection

The purpose of the *value* library is to make runtime value introspection available in OCaml. This library uses `camlp4` and `type-conv` to associate to any ML type definition annotated with the keyword `value` a pair of functions which will marshall/unmarshall any value of that type into a simpler and well-defined ML value, which is especially designed to make introspection easier. Hence, for a given type `t`, the library generates two functions `value_of_t : t -> Value.t` and `t_of_value : Value.t -> t`, where `Value.t` is defined as follows:

```
module Value = struct                                                OCaml
  type t =
    | Int of int64 | Bool of bool | Float of float | String of string
    | Arrow of string
    | Enum of t list
    | Tuple of t list
    | Dict of (string × t) list
    | Sum of string × t list
    | Null
    | Value of t
    | Ext of (string × int64) × t
    | Rec of (string × int64) × t
    | Var of (string × int64)
end
```

3.1 Basic Types

`Value.t` mimics the way ML values are represented in the heap and exposes that representation at runtime to the user. It defines a structure similar to the one defined by *type-of*. The differences are that the *value* library is building values instead of types, so for example the integer 42 is represented as `Int 42L`. Functional values are (un-)marshaled using the built-in OCaml `Marshal.{from_string,to_string}` functions. As for the *type-of* library, the basic values can be composed to form either a tuple with `Tuple`, a record or an object with `Dict` or a (polymorphic) variant with `Sum`.

For example, when the basic types defined in section 2.1 are annotated with the keyword `value`, they will generate ML expressions equivalent to:

```
(* Auto-generated code for "type tuple = ... with value" *)                    OCaml
let value_of_tuple (i,j) =
    Ext( ("tuple", 0), Tuple [ Int (Int64.of_int32 i); String j ] )

let tuple_of_value = function
    | Ext ( ("tuple", _), Tuple [ Int i; String j ]) → (Int64.to_int32 i, j)
    | _ → failwith "runtime error"

(* Auto-generated code for "type record = ... with value" *)
let value_of_record t =
    Ext ( ("record", 0), Dict [ "foo", String t.foo ] )

let record_of_value = function
    | Ext ( ("record", _), Dict d) when List.mem_assoc "foo" d → { foo = List.assoc "foo" d }
    | _ → failwith "runtime error"

(* Auto-generated code for "type variant = ... with value" *)
let value_of_variant = function
    | Foo → Ext ( ("variant", 0), Sum ("Foo", []) )
    | Bar b → Ext ( ("variant", 0), Sum ("Bar", [ Bool b ] ) )

let variant_of_value = function
    | Ext ( ("variant", _), Sum ( "Foo", []) ) → Foo
    | Ext ( ("variant", _), Sum ( "Bar", [ Bool b ] ) ) → Bar b
    | _ → failwith "runtime error"
```

3.2 Type Variables

Values whose type uses a type variable are built by induction on that type. As for the *type-of* library, the *value* library is easily extensible: for a type variable `t`, the user can choose to modify the type definition of `t` in order to add the keyword `value` and let the *value* library generate the `value_of_t` and `t_of_value` functions, or manually write these two functions.

Hence, in the following example, `value_of_x` and `x_of_value` might either be function generated from the definition of `t` or be user-defined:

```
(* User-defined datatype *)                                                    OCaml
type t = x option with value

(* Auto-generated code *)
let value_of_t = function
    | None → Ext (("t", 0), Null)
    | Some x → Ext (("t", 0), Value (value_of_x x))

let t_of_value = function
    | Ext(("t", _), Null) → None
    | Ext(("t", _), Value x) → Some (x_of_value x)
    | _ → failwith "runtime error"
```

3.3 Recursive Types

The first argument (of type `string * int64`) of type constructors `Ext`, `Rec` and `Var` is called a *variable*. Its first component (of type `string`) is the name of the type it comes from; the second one (of type `int64`) is an unique identifier which can be seen as the memory address in the heap of the corresponding value; however, this identifier is unique inside that value only so it cannot be used to do quick comparisons between two different values of type `Value.t`.

A value of type `Value.t` is *well-formed* if any `Var v` appears only in the scope of a recursive declaration; i.e. if we can find a upper-expression `Rec(v, e)` such that `Var v` is a free variable in `e`. In this case, `Var v` can be bound to the expression `e` and it can be substituted to it without changing the semantics of the term.

Furthermore, a value which has a recursive type does not necessary mean that value is recursive, the best example being tree-like structures. To make that distinction clear, a value `Rec(v,e)` means that `Var v` is free and *appears* in `e`; thus bounding `v` to `e` makes that value recursive. In the other hand, a value `Ext(v,e)` means that `Var v` does not appear in `e`.

For example, consider the following recursive type definition and the associated auto-generated code, where `free_vars : Value.t -> (string * int64) list` computes the free variables of an expression:

```
(* User-generated datatype *)
type t = { x : x } and x = { t : t} with value

(* Auto-generated code *)
let value_of_t, value_of_x =
    let rec value_of_t_aux xs ts t =
        if List.mem_assq t ts then
            Var ("t", List.assq t ts)
        else begin
            let id = new_id () in
            let res = Dict ( "x", value_of_x_aux xs ( (t,id) :: ts ) t.x )
                if List.mem (t,id) (free_vars res) then
                    Rec (("t",id), res)
                else
                    Ext (("t",id), res)
        end
    and value_of_x_aux xs ts x = ... in
    value_of_t_aux [] [], value_of_x_aux [] []
```

OCaml

We do not detail here how the functions `t_of_value` and `x_of_value` are generated, but the idea is quite similar: the algorithm needs to keep track of which ML value is associated to the unique identifiers stored into the `Rec` and `Var` constructors and reconstruct the ML value accordingly. It may also need to use some unsafe functions from the `Obj` module to initialize the induction.

4 SQL backend

We now show how to use the *type-of* and *value* libraries to build an integrated SQL backend to persist ML values. This backend is integrated seamlessly with OCaml: the user does not have to worry about writing SQL queries manually.

For each type definition `t` annotated with the keyword `orm`, a tuple of functions to persist and access the saved values are automatically generated:

```
(* User-defined datatype *)
type t = ... with orm

(* Auto-generated signatures *)
val t_init: string → (t, [ `RW ]) db
val t_init_read_only: string → (t, [ `RO ]) db
val t_get: (t, [< `RW | `RO ]) db → ... → t list
val t_save: (t, [ `RW ]) db → t → unit
val t_delete: (t, [ `RW ]) db → t → unit
```
OCaml

The `t_get` function has a part of its signature left unspecified; this is because the type of the query arguments are parameterized by `t` (see §5 for an example of query arguments). As an additional layer of type-safety, the database handle has a phantom polymorphic variant `[`RO|`RW]` that distinguishes between mutable and immutable database handles. This causes a compilation error if, for example, an attempt is made to delete a value to a read-only database.

4.1 Database Initialization

In this section, we explain how SQL schema can be derived using the *type-of* library. We define $\mathcal{N}$ as a collection of names closed under the following operations:

- $\bot \in \mathcal{N}$;
- $\texttt{string} \subset \mathcal{N}$;
- if $n \in \mathcal{N}$ and $i \in \mathbb{N}$ then $n_i \in \mathcal{N}$;
- if $n \in \mathcal{N}$ and $m \in \mathcal{N}$, then $n_m \in \mathcal{N}$.

The SQL schema syntax is then defined as follows, where $i \in \mathbb{N}$ and $n \in \mathcal{N}$:

$$\begin{aligned} \texttt{type} &::= I(i) \mid R \mid T \mid B \\ \texttt{field} &::= \langle n : \texttt{type} \rangle \\ \texttt{table} &::= n \vdash \{\texttt{field}, \ldots, \texttt{field}\} \end{aligned}$$

Such schema can be easily translated into an SQL query to create the appropriate tables in a database. First, `type` is translated into SQL with the following rule:

- $I(i) \rightarrow$ `INTEGER` if $i \leq 64$;
- $I(i) \rightarrow$ `TEXT` if $i > 64$;
- $R \rightarrow$ `REAL`;
- $T \rightarrow$ `TEXT`;
- $B \rightarrow$ `BLOB`.

Second, each value of the SQL schema domain corresponds naturally to an SQL query for creating a table in a given database. For example the schema $T \vdash \{\langle f_1 : R \rangle, \langle m_n : T \rangle\}$ can be intuitively associated to the following SQL query:

```
CREATE TABLE T
    (__id__ INTEGER PRIMARY KEY AUTOINCREMENT, f__1 REAL, m__n TEXT)
```
SQL

where `__id__` is an internal field which is used to uniquely identify each row in a given table and `f__1` and `m__n` are a possible translations of names f_1 and m_n to avoid name-clashes (we assume here that `__` is not used by the programmer).

We have shown how to derive an SQL query from an SQL schema. We now derive the SQL schema from an ML type `Type.t`, depicted in Figures 1 and 2. The purpose of this is to obtain consistent SQL tables in which a value of type `Value.t` can be stored naturally instead of manual conversion by the programmer.

$$
\begin{array}{ll}
(1) & [\![\, n, \texttt{Int}(i) \,]\!]_F = \{\, \langle n : I(i) \rangle \,\} \\
(2) & [\![\, n, \texttt{Float} \,]\!]_F = \{\, \langle n : R \rangle \,\} \\
(3) & [\![\, n, \texttt{String} \,]\!]_F = \{\, \langle n : T \rangle \,\} \\
(4) & [\![\, n, \texttt{Arrow \_} \,]\!]_F = \{\, \langle n : B \rangle \,\} \\
(5) & [\![\, n, \texttt{Option}\ t \,]\!]_F = \{\, \langle n_0 : I(1) \rangle \,\} \cup [\![\, n_0, t \,]\!]_F \\
(6) & [\![\, n, \texttt{Enum}\ t \,]\!]_F = \{\, \langle n : I(64) \rangle \,\} \\
(7) & [\![\, n, \texttt{Tuple}(\prod_{i \in [1..I]} t_i) \,]\!]_F = \bigcup_{i \in [1..I]} [\![\, n_i, t_i \,]\!]_F \\
(8) & [\![\, n, \texttt{Dict}(\prod_{i \in [1..I]} (m_i, t_i)) \,]\!]_F = \bigcup_{i \in [1..I]} [\![\, n_{m_i}, t_i \,]\!]_F \\
(9) & [\![\, n, \texttt{Sum}(\sum_{i \in [1..I]} (m_i, \prod_{j \in [1..J_i]} t_{i,j})) \,]\!]_F = \{\, \langle n_0 : T \rangle \,\} \bigcup_{i \in [1..I]} \bigcup_{j \in [1..J_i]} [\![\, n_{m_{ij}}, t_{i,j} \,]\!]_F \\
(10) & [\![\, n, \texttt{Ext}\ (\_, t) \,]\!]_F = \{\, \langle n : I(64) \rangle \,\} \\
(11) & [\![\, n, \texttt{Rec}\ (\_, t) \,]\!]_F = \{\, \langle n : I(64) \rangle \,\} \\
(12) & [\![\, n, \texttt{Var}\ (\_, t) \,]\!]_F = \{\, \langle n : I(64) \rangle \,\}
\end{array}
$$

Fig. 1. Field semantics

Figure 1 shows how to inductively build the collection of fields from a name and an element of `Type.t`, i.e. it defines a function $[\![.]\!]_F : \mathcal{N} \times \texttt{Type.t} \to \{\texttt{field}\}$. Equations (1)-(4) translates basic constructors of `Type.t` into simple fields with the appropriate type; equation (5) adds a new internal field to store if the value is set or not; moreover the current name is changed and that changed is propagated through the induction. Equations (6) and (10)-(12) means that enumeration and type variables are stored in separate tables and thus the row ID of this foreign table need to be stored in the current table. Finally, equation (7)-(9) fold the induction through the sub-terms of the current term of type `Type.t`, and propagate the name changes (i.e. we ensure that each sub-induction call has a different field name).

Figure 2 shows how to build the set of SQL tables from a name and element of `Type.t`, i.e. it defines a function $[\![.]\!]_T : \mathcal{N} \times \texttt{Type.t} \to \{\texttt{table}\}$. In equations (13)-(17), basic constructors of `Type.t` and option type do not affect the set of tables (they are handled in the previous field semantics). Equations (18)/(22)-(24) create foreign tables, which are referenced as `int64` integers from the field semantics in Equations (6)/(10)-(12). Moreover, equation (18) add the fields $\{\langle next : I(64) \rangle, \langle size : I(64) \rangle\}$ to the collections of fields computed by $[\![\cdot]\!]_F$: an enumeration is stored as a simply linked list in the database. Equations (19)-(21) also fold the induction through the current term as with the semantics in (7)-(9).

$$[\![n, \texttt{Int}(i)]\!]_T = \emptyset \tag{13}$$

$$[\![n, \texttt{Float}]\!]_T = \emptyset \tag{14}$$

$$[\![n, \texttt{String}]\!]_T = \emptyset \tag{15}$$

$$[\![n, \texttt{Arrow \_}]\!]_T = \emptyset \tag{16}$$

$$[\![n, \texttt{Option } t]\!]_T = \emptyset \tag{17}$$

$$[\![n, \texttt{Enum } t]\!]_T = [\![n_0, t]\!]_T \cup \{ n_0 \vdash \{\langle next : I(64)\rangle, \langle size : I(64)\rangle\} \cup [\![\bot, t]\!]_F \} \tag{18}$$

$$[\![n, \texttt{Tuple}(\prod_{i\in[1..I]} t_i)]\!]_T = \bigcup_{i\in[1..I]} [\![n_i, t_i]\!]_T \tag{19}$$

$$[\![n, \texttt{Dict}(\prod_{i\in[1..I]} (m_i, t_i))]\!]_T = \bigcup_{i\in[1..I]} [\![n_{m_i}, t_i]\!]_T \tag{20}$$

$$[\![n, \texttt{Sum}(\sum_{i\in[1..I]} (m_i, \prod_{j\in[1..J_i]} t_{i,j}))]\!]_T = \bigcup_{i\in[1..I]} \bigcup_{j\in[1..J_i]} [\![n_{m_{ij}}, t_{i,j}]\!]_T \tag{21}$$

$$[\![\_, \texttt{Ext } (n, t)]\!]_T = \{ n \vdash [\![\bot, t]\!]_F \} \cup [\![n, t]\!]_T \tag{22}$$

$$[\![\_, \texttt{Rec } (n, t)]\!]_T = \{ n \vdash [\![\bot, t]\!]_F \} \cup [\![n, t]\!]_T \tag{23}$$

$$[\![\_, \texttt{Var } (n, t)]\!]_T = \{ n \vdash [\![\bot, t]\!]_F \} \cup [\![n, t]\!]_T \tag{24}$$

Fig. 2. Table semantics

Finally, the table semantics of a type t can then be defined as $[\![\bot, t]\!]_T$.

At the OCaml interface layer, the initialization function connects to the database and dynamically confirms that it has the expected set of tables. If it creates a new database, it also saves the full `Type.t` in a special `__Types__` table for this purpose in the future. Some changes in the types between database initialization and use are safe due to the abstraction provided by *type-of*; records can be converted into objects, or polymorphic variants [3] can become normal ones. We are investigating automated database migration between differing schemas as a topic of future work.

5 Example: Photo Gallery

Due to space constraints, we do not explain the full semantics of queries and writes in this paper. Instead, we choose to illustrate the capabilities of the ORM library by constructing a simple photo gallery. We start that example by defining the basic ML types corresponding to a photo gallery:

```
type image = string
and gallery = {
    name: string;
    date: float;
    contents: image list;
} with orm
```
OCaml

We hold an `image` as a binary string, and a gallery is a named list of images. First, initializations functions are generated for both `image` and `gallery`:

```
val image_init : string → (image, [ `RW ]) db
val gallery_init : string → (gallery, [ `RW ]) db
val image_init_read_only : string → (image, [ `RO ]) db
val gallery_init_read_only : string → (gallery, [ `RO ]) db
```
OCaml

Intuitively, calling `gallery_init` will:
(i) use `type-of` to translate the type definitions into:

```
let type_of_image = Ext ( "image", String )
let type_of_gallery =
    Ext("gallery", Dict [("name", String); ("date", Float) ; ("contents", Enum type_of_image)])
```
OCaml

(ii) use the rules defined by Figures 1 and 2 to generate the database schema:

```
CREATE TABLE image (__id__ INTEGER PRIMARY KEY, __contents__ TEXT);
CREATE TABLE gallery (__id__ INTEGER PRIMARY KEY, date REAL, contents INTEGER);
CREATE TABLE gallery__contents (__id__ INTEGER PRIMARY KEY,
    __next__ INTEGER, __size__ INTEGER, __contents__ INTEGER);
```
SQL

Second, using the `value` library, any value of type `image` or `gallery` can be translated into a value of type `Value.t`. Using rules similar to the ones defined in Figures 1 and 2, saving functions can be then defined, having as signature:

```
val image_save : (image, [ `RW ]) db → image → unit
val gallery_save : (gallery, [ `RW ]) db → gallery → unit
```
OCaml

Finally, using `type-of`, functions to access the database are generated, with the following signature:

```
val image_get : (image, [< `RO | `RW ]) db →
    ?value:[`Contains of string | `Eq of string] ] →
    ?custom:(image → bool) →
    image list
val gallery_get : (gallery, [< `RO | `RW ]) db →
    ?name:[ `Eq string | `Contains string] →
    ?date:[ `Le float | `Ge float | `Eq float | `Neq float] →
    ?custom:(gallery → bool) →
    gallery list
```
OCaml

For both types, we are generating: (i) arguments that can be easily translated into an optimized SQL queries; and (ii) a more general (and thus slow) custom query function directly written in OCaml. On one hand, (i) is achieved by generating optional labelled arguments with the OCaml type corresponding to the fields defined by Figure 1. This allows the programmer to specify a conjunction of type-safe constraints for his queries. For example, the field `name` is of type string which is associated to the constraint of type `[ 'Eq of string | 'Contains of string ]`. Values of this type can then be mapped to SQL equality or the `LIKE` operator. On the other hand, (ii) is achieved using a SQLite extension to define custom SQL functions—in our case we register an OCaml callback directly. This is relatively slow as it bypasse the query optimizer, but allows the programmer to define very complex queries.

```
let db = gallery_init "my_gallery.db" in
let i = new_image () in
let gallery = { name="WTG2010"; date=today(); contents=[i] } in
gallery_save db gallery;
match gallery_get  name:(Eq "WTG2010") db with
    | [ g ] → printf "Found 1 gallery: %s" g.name
    | _ → failwith "Wrong numver of galleries"
```
OCaml

The above code snippet saves a gallery named "WGT2010" containing an unique fresh image in a database called `my_gallery.db`. It then queries all the galleries whose name is strictly equal to "WGT2010". It expects to find exactly one gallery with this name; otherwise it throws an error.

6 Related Work and Conclusions

There are a number of extensions to functional languages to enable general meta-programming, such as Template Haskell [9] and MetaOCaml [10]. MetaHDBC [1] uses Template Haskell to connect to a database at compile-time and generate code from the schema; in contrast, we derive schemas directly from types in order to make the use of persistence more integrated with existing code. We avoid a dependency on MetaOCaml by using `camlp4` in order to fully use the OCaml toolchain (particularly ARM and AMD64 native code output), and also because we only need a lightweight syntax extension instead of full meta-programming support. We believe that our work is simpler and easier to extend than Yallop's *deriving* [11] which is inspired by the construct in the same name in Haskell [4]. Language-integrated constructs to manipulate databases is also an active topics for mainstream languages, such as the LINQ [7] library for the .NET framework. The small syntax extension we are proposing in this paper is more naturally integrated with the host language.

We have shown how a type and value introspection layer using the AST transformer built into OCaml can be used to create useful persistence extensions for the language that does not require manual translation. As future work, we are building libraries for network and parallel computation using the same base libraries. The library is open-source and available at: `http://github.com/mirage/orm`.

References

[1] Metahdbc, 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[3] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[4] S. L. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[5] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, 2005.

[6] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. *SIGOPS Oper. Syst. Rev.*, 41(3):101–114, 2007.

[7] E. Meijer, B. Beckman, and G. M. Bierman. Linq: reconciling object, relations and xml in the .net framework. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, page 706. ACM, 2006.

[8] M. Mottl. type-conv: a library for composing automated type conversions in ocaml, 2009.

[9] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[10] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50, Dagstuhl Castle, Germany, March 2004. Springer.

[11] J. Yallop. Practical generic programming in ocaml. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 83–94, New York, NY, USA, 2007. ACM.