Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 264 (5) (2011) 3–21

www.elsevier.com/locate/entcs

Dynamics for ML using Meta-Programming

Thomas Gazagnaire

INRIA Sophia Antipolis-Méditerranée,
2004 Route des Lucioles,
BP 93, 06902 Sophia Antipolis Cedex, France
thomas.gazagnaire@inria.fr

Anil Madhavapeddy

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

Abstract

We present the design and implementation of dynamic type and value introspection for the OCaml language. Unlike previous attempts, we do not modify the core compiler or type-checker, and instead use the `camlp4` metaprogramming tool to generate appropriate definitions at compilation time. Our dynamics library significantly eases the task of generating generic persistence and I/O functions in OCaml programs, without requiring the full complexity of fully-staged systems such as MetaOCaml. As a motivating use of the library, we describe a SQL backend which generates type-safe functions to persist and retrieve values from a relational database, without requiring programmers to ever use SQL directly.

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

1 Introduction

One of the great advantages of programming languages inheriting the Hindley-Milner type system [6,17] such as OCaml [12] or Haskell [11] is the conciseness and expressiveness of their type language. For example, sum types in these languages are very natural to express and use when coupled with pattern-matching. These concepts can be translated to C or Java, but at the price of a costly and unnatural encoding. Parameterised types and mutually recursive types can also be used in OCaml or Haskell to let the user define arbitrary complex data-types: part of the art of programming in such languages is to encode invariants of the problem being resolved into the types, and let the compiler statically ensure these invariants are met during the whole execution of the program.

Historically, languages implementing such type systems are statically typed, with safety enforced at compile-time and details of types forgotten at runtime. This helps

doi:10.1016/j.entcs.2011.06.002

generate efficient code with compact runtime support [9,15]. The lack of runtime type introspection does make some tasks more difficult to perform than with more dynamically-typed languages such as Python or Java. Pretty-printing, conversions between different types, or value persistence is a largely manual process in ML-like languages, and can be tedious and error-prone. Haskell solves these problems by using *type-classes* [4,22], which is a natural concept but difficult to implement. The type-inference algorithm becomes more complex and the runtime implementation suffers some performance penalties. In this paper, we concentrate on dynamic types, but our work is influenced by ideas coming from type-classes.

Dynamic typing in the context of statically-typed programming languages had been extensively studied [1] and even implemented in early versions of the OCaml compiler. [1] Such solutions involve so-called objects with dynamic types (shortened to *dynamics*), which consist of pairing a value v with a type expression τ such that v is of type τ. The compiler is modified to:

- add a built-in type `dyn` such that all dynamics (v, τ) are of type `dyn`;
- add two constructs to communicate between type `dyn` and other types: one to pair any value with its static type, and one to check if a dynamic value is of type τ, and if so let the programmer read the associated value.

Such constructions are very powerful but difficult to implement correctly when combined with a rich type environment. Moreover, their implementation is quite intrusive in the compiler source code, as they modify the host type-system and language constructs. Possibly as a result of this complexity, modern versions of OCaml no longer have dynamics as a language feature.

In this paper, we describe a simplified implementation of dynamics in OCaml, based on *staged programming* to generate and execute code fragments as part of the compilation process [21]. We describe a 2-stage transformer that is sufficient for generating information about dynamic types, and we illustrate the use of that information to show how to build a storage layer which can easily persist ML values.

A key benefit of our approach is that it does not need to modify the core OCaml compiler, and instead uses the `camlp4` AST transformer to generate extra code at compilation time. Our implementation: (*i*) parses a large subset of OCaml types to a more succinct and expressive form than the syntax tree which `camlp4` provides; (*ii*) implements an Object-Relational Mapping (ORM) which defines an efficient conversion to and from SQL; and (*iii*) provides 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. One of the benefits of our approach is that it works as a library to the standard OCaml distribution—no modifications to the OCaml tool-chain are needed. For example:

[1] In OCaml 2.x in the `dynamics` branch in source control. [13]

```
type t = { name: string; mail: string } with orm                    OCaml
let authors =
    [ { name="Anil"; mail="avsm2@cam.ac.uk" };
      { name="Thomas"; mail="tgazagna@inria.fr" } ]
let main () =
    let db = t_open "contacts" in
    t_save db authors;
    let cam = t_get ~mail:('Contains "ac.uk") db in
    printf "Found %d @ac.uk" (List.length cam)
```

The type `t` is a standard OCaml type, with an annotation to mark it as a storage type. Variables of type `t` can be saved and queried via the `t_open`, `t_save` and `t_get` functions. The backend uses the SQLite database library, and SQL is automatically generated from the applications datatypes and never used by the programmer directly.

Parts of the extension were developed for use in XenServer and the Xen Cloud Platform [19]—a large, complex OCaml application that has been developed since 2006. The Xen Cloud Platform runs in an embedded Linux distribution and controls virtual machine activity across large pools of physical machines, and so requires efficient and reliable storage and I/O.

In the remainder of the paper, we first describe the type parsing (§2) and value introspection libraries (§3). Then we motivate its use by illustrating the design of a SQL persistence layer for ML values (§4), and finally an example of a simple photo gallery (Section §5).

2 Type Introspection

2.1 Formal background

First of all, let us focus on the (declarative) type language of ML. Let us consider two disjoint sets of names $\mathcal{R}$ and $\mathcal{A}$. We consider a type definition to be an equation $\rho(\hat{\alpha}) = t$ where $\rho \in \mathcal{R}$ is a type variable, $\hat{\alpha}$ is a possibly empty collection $\{\alpha_1, \ldots, \alpha_n\}$ of type parameters $\alpha_i \in \mathcal{A}$ and t is described by the following syntax :

$\mathtt{tt} ::= \mathtt{base}$		a base type
$\mid \langle n_1 : \mathtt{tt}\rangle_{M_1} \times \ldots \times \langle n_k : \mathtt{tt}\rangle_{M_k}$	$n_i \in \mathcal{N}, M_i \in \{\cdot, \mathtt{M}\}$	product type
$\mid \langle n_1 : \mathtt{tt}\rangle + \ldots + \langle n_k : \mathtt{tt}\rangle$	$\forall i, n_i \in \mathcal{N}$	sum type
$\mid [\mathtt{tt}]$		enumeration type
$\mid \mathtt{tt} \to \mathtt{tt}$		function type
$\mid \rho \mid \rho(\mathtt{tt}, \ldots, \mathtt{tt})$	$\rho \in \mathcal{R}$	type variable
$\mid \alpha$	$\alpha \in \mathcal{A}$	type parameter

$\mathtt{base} ::= \mathtt{UNIT} \mid \mathtt{INT}(\mathbb{N}) \mid \mathtt{FLOAT} \mid \mathtt{STRING}$

Basic types correspond to all the basic types that can be defined in OCaml. The $\mathtt{INT}(i)$ constructor stands for an i-bit integer. Names of named product and sum come from an infinite set of symbols $\mathcal{N}$. The parameters M_i in the named product indicate that such fields can be mutable; we will write $\langle n : t\rangle$ when a field

is immutable, and $\langle n : t \rangle_M$ otherwise. The difference between a named product and an enumeration is that values of type enumeration are unbound and so there is no way to statically determine their size n; however, this bound is explicit for a named product. Cartesian products (or *tuples*) can be naturally encoded into a named product by giving to fields the names corresponding to their position in the tuple:

$$t_1 \times \ldots \times t_n \stackrel{\texttt{def}}{=} \langle 1 : t_1 \rangle \times \ldots \times \langle n : t_n \rangle$$

$\rho(t_1, \ldots t_n)$ is the total application to ρ of its type parameters: arity consistency is not a problem as type functions are always total in ML. Finally, when $\hat{\alpha} = \emptyset$, we write the type definition as $\rho = t$ and we say that t is a *monomorphic* type.

Now, let us consider ML programs from a type perspective, by ignoring values and considering only types declarations. In the *absence of recursive modules* and by flattening the name-space of types, every ML program is a sequence of recursive type declarations. A program P can be modeled as follows:

$$P = \left\{ \begin{array}{c} \rho_{1,1}(\hat{\alpha}_{1,1}) = t_{1,1} \\ \vdots \\ \rho_{1,n_1}(\hat{\alpha}_{1,n_1}) = t_{1,n_1} \end{array} \right. \cdot \ldots \cdot \left\{ \begin{array}{c} \rho_{1,k}(\hat{\alpha}_{k,1}) = t_{k,1} \\ \vdots \\ \rho_{k,n_k}(\hat{\alpha}_{k,n_k}) = t_{k,n_k} \end{array} \right.$$

where $\rho_{i,j}$ can only appear in the term $t_{k,l}$ if either $i < k$ or $k = i$ and $l \leq n_i$. Moreover, any type parameter α appearing in a $t_{i,j}$ term should also be a member of the corresponding $\hat{\alpha}_{i,j}$ on the left-hand side of the equation definition.

The main goal of type introspection is to give an intuitive and easy-to-use run-time representation of the types manipulated by the program. We believe than an equational representation of the types, even if it is compact and intuitive to write, is not easy to *use* from a programmer's perspective. Moreover, using such equations requires a dynamic context which binds previous type variables to type expressions, and this is impossible to have at preprocessing time. [2] Instead, we expose a fix-point representation obtained by unfolding the types variables within the same recursive set of equations and an inductive call for previously defined type variables: this representation is finite and computable at preprocessing time, while preserving the same type structures that the programmer has defined. The main restriction is that some advanced uses of the module system, such as recursive module definitions, cannot be expressed using this technique.

2.2 Fixed-point Type Declarations

We now explain how to incrementally transform the sequence of recursive equations into a sequence of independent fix-point declarations, where the extended type structure (i.e. including abbreviation definitions) of the types is not lost. We first

[2] Recall that one of our implementation constraints is to obtain dynamics using the `camlp4` preprocessor instead of compiler modifications.

extend the syntax for types defined above with a new fix-point constructor:

$$\texttt{tt} ::= \ldots \mid \mathit{type}\ \rho \cdot \texttt{tt}$$

We then say that a type expression of the form $\mathit{type}\ \rho \cdot t$ is recursive if the type variable ρ is a free variable in the type expression t; this is a static property, which will be denoted by $\mathit{type}_R\ \rho \cdot t$ and is equivalent to the μ construct in type theory.

Furthermore, such a type expression is mutable if the symbol M appears in the type expression t. This can also be statically decided, and is denoted by $\mathit{type}_M\ \rho \cdot t$. Both of these static properties can be composed, so one can have a mutable and recursive expression which will be denoted by $\mathit{type}_{RM}\ \rho \cdot t$.

Let us now define how to translate from a sequence of recursive equations into a sequence of fix-points expressions, while preserving some kind of structure that the programmer would expect. This problem had been studied in the context of structural type-equivalence or subtyping [2,3,5,8] and the algorithm used is based on a gaussian elimination technique; the correctness on that technique is ensured by Bekić's Theorem which states that any mutually recursive types can always be defined as simple μ-types [23].

We have adapted this algorithm to work in our setting—namely, for mutually recursive parametrised types. First, we define $t[u/\mu(\hat{\alpha})]$ as the substitution of the type variable μ (and its possibly empty type parameters $\hat{\alpha}$) by the expression u in t. This operation is defined by induction on t :

$$\begin{aligned}
\text{if } b \in \texttt{base} : b[u/\mu(\hat{\alpha})] &= b \\
\big(\langle n_1 : t_1\rangle \times \ldots \times \langle n_k : t_k\rangle\big)[u/\mu(\hat{\alpha})] &= \langle n_1 : t_1[u/\mu(\hat{\alpha})]\rangle \times \ldots \times \langle n_k : t_k[u/\mu(\hat{\alpha})]\rangle \\
\big(\langle n_1 : t_1\rangle + \ldots + \langle n_k : t_k\rangle\big)[u/\mu(\hat{\alpha})] &= \langle n_1 : t_1[u/\mu(\hat{\alpha})]\rangle + \ldots + \langle n_k : t_k[u/\mu(\hat{\alpha})]\rangle \\
\big[t\big][u/\mu(\hat{\alpha})] &= \big[t[u/\mu(\hat{\alpha})]\big] \\
\big(t_1 \to t_2\big)[u/\mu(\hat{\alpha})] &= t_1[u/\mu(\hat{\alpha})] \to t_2[u/\mu(\hat{\alpha})] \\
\alpha[u/\mu(\hat{\alpha})] &= \alpha
\end{aligned}$$

For the substitution of type variables, we need to consider two cases. First, if $\rho \neq \mu$ then, the induction is trivial:

$$\begin{aligned}
\rho[u/\mu(\hat{\alpha})] &= \rho \\
\rho(t_1, \ldots, t_n)[u/\mu(\hat{\alpha})] &= \rho\big(t_1[u/\mu(\hat{\alpha})], \ldots, t_n[u/\mu(\hat{\alpha})]\big)
\end{aligned}$$

However, when $\rho = \mu$, type arity has also to match and then we have :

$$\begin{aligned}
\rho[u/\rho] &= u \\
\rho(t_1, \ldots, t_n)[u/\rho(\hat{\alpha})] &= u\Big[t_1[u/\rho(\hat{\alpha})]\Big/\alpha_1\Big] \ldots \Big[t_n[u/\rho(\hat{\alpha})]\Big/\alpha_n\Big]
\end{aligned}$$

In the first equation, $\hat{\alpha} = \emptyset$, and it is not possible to substitute a monomorphic type by a polymorphic one. In the second equation, $\hat{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$ where n is the same as in $t_1 \ldots t_n$ and $u[v_1/\alpha_1] \ldots [v_n/\alpha_n]$ is left-associative, i.e.

$\big(\ldots(u[v_1/\alpha_1])\ldots\big)[v_n/\alpha_n]$; this corresponds to first doing induction on the arguments, and then substituting in the expression of the type parameters with the corresponding computed arguments.

Let us now consider a program P, viewed as a sequence of recursive equations, and let us focus on the last equation system of that sequence :

$$P : X_1 \cdot \ldots \cdot X_{n-1} \cdot \begin{cases} \rho_1(\hat{\alpha}_1) = t_1 \\ \vdots \\ \rho_n(\hat{\alpha}_n) = t_n \end{cases}$$

We want to associate to P a sequence $\underline{P}$ of fix-point instructions of the form :

$$\underline{P} : \underline{X}_1 \cdot \ldots \cdot \underline{X}_{n-1} \cdot \begin{cases} \underline{\rho}_1(\hat{\alpha}_1) \leftarrow \varphi(\rho_1) \\ \vdots \\ \underline{\rho}_n(\hat{\alpha}_n) \leftarrow \varphi(\rho_n) \end{cases}$$

Γ denotes the mapping associating each ρ_i to the corresponding t_i in the last equation system of P. Also, $dom(\Gamma)$ is the domain of Γ, i.e. the collection $\{\rho_1, \ldots, \rho_n\}$. In order to define, ϕ, we first introduce an intermediate function called $\varphi^\star$. This function inductively unfolds a type expression by replacing each type variable by its value exactly once; in order to do so, it uses a set of type variables to remember the ones already unfolded. Hence, the signature of $\varphi^\star$ is :

$$\varphi^\star : 2^{dom(\Gamma)} \times dom(\Gamma) \to \mathtt{tt}$$

It associates a fix-point type expression to a collection of type variables (the variables already unfolded) and a type variable (the variable to be unfolded) as follows :

$$\varphi^\star_R(\rho) = \begin{cases} \rho & \text{if } \rho \in R \\ \underline{\rho} & \text{if } \rho \notin dom(\Gamma) \\ type\ \rho \cdot \Gamma(\rho)\Big[\varphi^\star_{R_\rho}(\rho_1)\Big/\rho_1(\hat{\alpha}_1)\Big] \ldots \Big[\varphi^\star_{R_\rho}(\rho_n)\Big/\rho_n(\hat{\alpha}_n)\Big] & R_\rho = \{\rho\} \cup R \end{cases}$$

Finally, φ is defined as :

$$\varphi(\rho) = \varphi^\star_\emptyset(\rho)$$

Intuitively, φ substitutes all the type variables by either their value if they are defined in the same set of recursive equations or by the inductive value computed previously otherwise, until all the variables in the right-hand side expression of the equality are bound. The only point of discussion is whether the type abbrevations should be preserved by this transformation; We really want to emphasize here that sometimes *it is not enough to preserve structural equivalence*. Indeed, the programmer may also wants to be aware of some memory structures, the best

example being when trying to persist values, as we will discuss more in depth in §4. A simple example can be described by the two programs P_1 and P_2 :

$$P_1 : \begin{cases} t_1 = \langle n : x \rangle \\ x = \texttt{INT}(31) \times \texttt{STRING} \end{cases} \qquad P_2 : \Big(t_2 = \langle n : \texttt{INT}(31) \times \texttt{STRING} \rangle \Big)$$

Here, t_1 and t_2 are structurally equivalent; however, our "programmer" intuition is that these two types should result in distinct relational schemas, where the table associated to t_1 will feature an explicit indirection to the table associated to x:

table t_1

id	n
	x(id)

table x

id	1	2
×		

table t_2

id	x_1	x_2

In this example, the table associated to type t_1 has two columns: one is a unique identifier; and the other column corresponding to field n contains identifiers referencing elements in the table associated to type x. The table associated to type x has three columns: the unique identifiers that column n in table t_1 is referencing, and the two others columns contain 31-bits integers and strings.

On the right hand side, the table associated to type t_2 has three columns, the columns associated to x being inlined directly. If we use a transformation based on structural equivalence to pass from the types to the schemas, the schema representation of t_1 and t_2 will be indistinguishable. Both tables will be similar to the one associated to t_2, which is not expected from a programmer's point-of-view.

The type system of ML is based on structural equivalence, and so our choice might seem contradictory as we distinguish types that are indistinguishable by the type system. However, in our practical experience [19], such indirections are always put in the code for some reason (to be able to use physical equality or to ensure maximum sharing for example) and so we believe that this extended structure should be pushed down to the persistence layer as well.

When applying φ on t_1, x and t_2 to obtain $\underline{P}_1$ and $\underline{P}_2$, we can remark that $\underline{t}_1$ and $\underline{t}_2$ are different (even using α-conversion) :

$$\underline{P}_1 : \begin{cases} \underline{t}_1 \leftarrow \mathit{type}\ t_1 \cdot \langle n : \mathit{type}\ x \cdot (\texttt{INT}(31) \times \texttt{STRING}) \rangle \\ \underline{x} \leftarrow \mathit{type}\ x \cdot (\texttt{INT}(31) \times \texttt{STRING}) \end{cases}$$
$$\underline{P}_2 : \Big(\underline{t}_2 \leftarrow \mathit{type}\ t_2 \cdot \langle n : \texttt{INT}(31) \times \texttt{STRING} \rangle \Big)$$

We now focus on a more complex example using recursion and inductive definitions. Consider a program P, with the sequence of recursive equations :

$$P : \Big(t = \texttt{STRING} \Big) \cdot \begin{cases} x = y(t) \\ y(\alpha) = [x \times \alpha] \end{cases}$$

We can now apply φ and check for static properties to obtain :

$$\underline{P} : \left(\underline{t} \leftarrow type\ t \cdot \texttt{STRING}\right) \cdot \begin{cases} \underline{x} \leftarrow type_R\ x \cdot \left(type\ y \cdot [x \times \underline{t}]\right) \\ \underline{y}(\alpha) \leftarrow type_R\ y \cdot \left[type\ x \cdot (y(\underline{t}) \times \alpha)\right] \end{cases}$$

Again, the obtained result is finite and preserves the extended type structure (type abbreviations and type structure). The usual solution [3] would have been to forget non-recursive *type* constructors to obtain the more compact but less precise:

$$\underline{P} : \left(\underline{t} \leftarrow \texttt{STRING}\right) \cdot \begin{cases} \underline{x} \leftarrow type_R\ x \cdot [x \times \underline{t}] \\ \underline{y}(\alpha) \leftarrow type_R\ y \cdot [y(\underline{t}) \times \alpha] \end{cases}$$

This more compact structure can trivially be obtained from our extended structure at a later stage.

2.3 Implementation

The transformation described in §2.2 has been implemented in OCaml as a preprocessing library, called *type-of*. This library uses `camlp4` and the `type-conv` framework [18] to make type introspection available to the programmer. Hence, for each type definition annotated with the special keyword `type_of`, it will automatically generate a *finite* value of type `Type.t` describing the type shape:

OCaml
```
module Type = struct
  type t =
    | Unit | Int of int option | Float | String
    | Dict of (string × ['RW|'RO] × t) list
    | Sum of (string × t list) list
    | Enum of t
    | Arrow of t × t
    | Var of string × (t list)
    | Param of string
    | Type of elt
  and elt = {
    recursive : bool;
    read_only : bool;
    name : string;
    contents : t }
end
```

As described previously, this transformation works for a large subset of ML types, including recursive and polymorphic types. We have the following correspondence: on the left-hand side, types as they were defined in §2.1; on the right-hand side,

[3] Recall we did not choose this method because it forgets information that the programmer explicitly annotated in their source code.

types as they are defined in an OCaml program:

$$t = \ldots \qquad \texttt{type t} = \ldots \texttt{ with type\_of}$$
$$\underline{t} \leftarrow \ldots \qquad \texttt{let type\_of\_t} = \ldots$$

$$t(\alpha_1, \ldots, \alpha_n) = \ldots \qquad \texttt{type } (\texttt{`a1}, \ldots, \texttt{`an}) \texttt{ t} = \ldots$$
$$\underline{t}(\alpha_1, \ldots, \alpha_n) \leftarrow \ldots \qquad \texttt{let type\_of\_t type\_of\_a1 ... type\_of\_an} = \ldots$$

The translation from a set of recursive equation into fix-point expressions is well-known. Our contributions are to: (i) make the fix-point expressions available to the programmer to inspect static types at run-time; and (ii) tailor the technique for preprocessing time using only syntactic information and keeping the core compiler tool-chain significantly simpler. Furthermore, modularity and abstraction are handled quite naturally using induction on types variables—a programming style close to the one used when type-classes are available. For example :

OCaml

```
(* type definition *)
type α t = A of α | X of x list
(* auto-generated code *)
let type_of_t type_of_a =
    Type {
        recursive : (is_recursive type_of_a) || (is_recursive type_of_x);
        read_only = (is_read_only type_of_a) && (is_read_only type_of_x);
        name = "t";
        contents = Sum [
            ( "A", [type_of_a] );
            ( "X", [Enum (type_of_x)] )
        ]
    }
```

In this case, `type_of_x` has to be defined previously for the program to compile. This definition may have either been automatically generated previously using the *type-of* library, or been defined previously by the user. The latter option makes the *type-of* library easily extensible, especially for abstract types.

3 Value Introspection

3.1 Formal background

As we did for types, we now introduce the syntax for values. The considered values are concrete memory representations; we thus define a collection $\mathcal{L}$ of memory locations and we assume that we have a memory function $\mathcal{M} : (\mathcal{L} \times \texttt{tt}) \rightarrow \texttt{vv}$ associating typed memory locations to values (implicitly performing a conversion from ML values into introspectable values), where introspectable values are described by

the following syntax :

$\texttt{vv} ::=$	$\texttt{base}$		a base value
	$\mid (\texttt{vv}, \ldots, \texttt{vv})$		tuple construction
	$\mid \langle n : \texttt{vv} \rangle$	$n \in \mathcal{N}$	sum construction
	$\mid (\gamma : \underline{\rho})$	$\gamma \in \mathcal{L}, \underline{\rho} \in \texttt{tt}$	typed variable
	$\mid \top$		an unknown value
$\texttt{base} ::=$	$\texttt{UNIT}$		no value
	$\mid \texttt{INT}(\mathbb{Z})$		integer constants
	$\mid \texttt{FLOAT}(\mathbb{R})$		real numbers
	$\mid \texttt{STRING}(\Sigma^\star)$		strings constants

Base values are tagged with their types; this can be performed directly when calling $\mathcal{M}$ as the type representation $\underline{\rho}$ as computed in §2.2 is available. For example, the integer 42 will hence be represented as $\texttt{INT}(42)$. We ensure that the type of such variable does not have any free parameters; it is not possible using our scheme to have a value representing an α-list. Furthermore, unlike the type syntax, value representations do not carry any names. This information is already present in the type description obtained earlier (see §2.2). Hence, programmers can reason by induction both on value and type runtime representations at the same time

The only constructs are unbounded product and sum constructors. Values corresponding to named sum types are built by remembering the name of the tag and the corresponding value. Functional values have no (explicit) runtime representation and are represented by the symbol $\top$.

As for types, our goal is to provide at runtime to the programmer a finite and easy-to-use representation of the values. However, unlike types, values are built and modified at runtime. It is thus impossible to build a translation at preprocessing time, as the program needs to run to actually produce values. We solve this by generating at preprocessing time, a pair of functions $\overrightarrow{\rho}: \texttt{ML} \to \texttt{vv}$ and $\overleftarrow{\rho}: \texttt{vv} \to \texttt{ML}$. These functions transform back and forth, at runtime, any ML value of type ρ into a finite representation whose syntax is $\texttt{vv}$, extended with a fix-point operator:

$$\texttt{vv} ::= \ldots \mid \mathit{val}\ \gamma \cdot \texttt{vv}$$

The extra fix-point operator is used to deal with cyclic values. Recursive types do not automatically imply corresponding cyclic values, so this information is not already encoded in the type description and must be explicitly encoded in the value. We denote by $FV(v)$ the free variables in v. We can easily define substitution on values; if u and v are two values and γ a variable, then $u[v/\gamma]$ is the value u where all instances of γ have been replaced by v :

$$\begin{aligned}
\big(u_1, \ldots, u_n\big)[v/\gamma] &= \big(u_1[v/\gamma] \times \ldots \times u_n[v/\gamma]\big) \\
\langle n : u\rangle[v/\gamma] &= \langle n : u[v/\gamma]\rangle \\
\gamma[v/\gamma] &= v \\
\text{Otherwise: } u[v/\gamma] &= u
\end{aligned}$$

We now detail how the functions are generated at preprocessing time. Recall that a program, from a type perspective, is a sequence of recursive equations :

$$P : X_1 \cdot \ldots \cdot X_{n-1} \cdot \begin{cases} \rho_1(\hat{\alpha}_1) = t_1 \\ \vdots \\ \rho_n(\hat{\alpha}_n) = t_n \end{cases}$$

Following the technique described in §2.2, we associate to that program a sequence of values $\overrightarrow{P}$:

$$\overrightarrow{P} : \overrightarrow{X}_1 \cdot \ldots \cdot \overrightarrow{X}_{n-1} \cdot \begin{cases} \overrightarrow{\rho}_1\ (\hat{\alpha}_1) = \psi(\underline{\rho}_1) \\ \vdots \\ \overrightarrow{\rho}_n\ (\hat{\alpha}_n) = \psi(\underline{\rho}_n) \end{cases}$$

such that :

$$\psi(\underline{\rho}) = (\psi^\star_\emptyset(\underline{\rho}) : \underline{\rho}); \text{ and}$$

$$\psi^\star_L(\underline{\rho})(\gamma) = \begin{cases} (\gamma : \underline{\rho}) & \text{if } \gamma \in L \\ val\ \gamma \cdot v[u_1/\gamma_1] \ldots [u_n/\gamma_n] & \text{if } \gamma \in FV(v[u_1/\gamma_1] \ldots [u_n/\gamma_n]) \\ v[u_1/\gamma_1] \ldots [u_n/\gamma_n] & \text{otherwise, where :} \\ & \bullet\ v = \mathcal{M}(\gamma, \underline{\rho}); \\ & \bullet\ \{\gamma_1, \ldots, \gamma_n\} = FV(v); \\ & \bullet\ \text{and } \forall i, u_i = \psi^\star_{\big(\{\gamma\}\ \cup\ L\big)}(\underline{\rho}_i)(\gamma_i). \end{cases}$$

As with types, the transformation of an implicit collection of recursive equations into a fix-point representation is done by induction. It suffices to substitute value locations by their contents and stop when all value variables are bound to an inner *val* declaration. It is worth emphasising that, even if they look similar, the function $\varphi^\star$ (defined in §2.2) and $\psi^\star$ are different. $\varphi^\star$ is a value computed at preprocessing time, and the substitutions are done only once; whereas $\psi^\star$ is a function that computes a new value each time it is called. This implies that every translation from an ML value into an element of `vv` is an expensive operation: all the memory of

the ML values needs to be scanned and some re-allocated into the new structure [4]. These costs are difficult to eliminate entirely, but our implementation (§3.2) uses lazy evaluation to evaluate only necessary parts of the translated ML value.

Using similar techniques, we also compute ψ^{-1} to get :

$$\overleftarrow{P} : \overleftarrow{X}_1 \cdot \ldots \cdot \overleftarrow{X}_{n-1} \cdot \begin{cases} \overleftarrow{\rho}_1\ (\hat{\alpha}_1) = \psi^{-1}(t_1) \\ \vdots \\ \overleftarrow{\rho}_n\ (\hat{\alpha}_n) = \psi^{-1}(t_n) \end{cases}$$

The only notable difference is, as ψ is not surjective, that ψ^{-1} can produce an exception if the dynamic value cannot be converted back to a normal ML value.

Let us now consider an example. We are assuming the memory function to have the following shape :

$$\mathcal{M} : \begin{cases} (x_1, \underline{x}) \rightarrow y \times \texttt{INT}(32) \\ (x_2, \underline{x}) \rightarrow y \times \texttt{INT}(52) \\ (y, \underline{y}) \rightarrow x_1 \times x_2 \times x_1 \end{cases}$$

with :

$$\underline{x} = type_R\ x \cdot \Big(\big(type\ y \cdot (x \times x \times x) \big) \times \texttt{INT}(31) \Big)$$
$$\underline{y} = type_R\ y \cdot \Big(type\ x \cdot \big(y \times \texttt{INT32} \big) \times type\ x \cdot \big(y \times \texttt{INT32} \big) \times type\ x \cdot \big(y \times \texttt{INT32} \big) \Big)$$

Then the runtime representation $\overrightarrow{x}\ (x_1) = \psi(\underline{x})(x_1)$ is the pair $(\gamma : \underline{\rho})$ where :

$$\gamma = val\ x_1 \cdot \Big(val\ y \cdot \big((x_1 : \underline{x}) \times (y \times \texttt{INT}(52)) \times (x_1 : \underline{x}) \big) \times \texttt{INT}(32) \Big)$$

3.2 Implementation

The transformation described in §3.1 has also been implemented as a library called *value*. As for the *type-of* library, it uses `camlp4` and `type-conv` to generate at preprocessing time, a pair of functions to translate to and from an ML value and a dynamic value expression. To this end, the ML value should be of an explicitly declared ML type, annotated with the keyword `value`. The dynamic value expression implements the syntax given in 3.1 :

[4] Some potentially large primitive values, such as `string` values will not be reallocated but passed by reference.

```
module Value = struct                                                    OCaml
    type loc
    type elt =
        | Unit | Int of int64 | Float of float | String of string
        | Tuple of elt list
        | Sum of string × elt
        | Unknown
        | Val of loc × elt
        | Var of loc
    type t = { val : elt; typ : Type.t }
end
```

We then have the following correspondence between the notations given in the previous section and the *value* library :

$$\overrightarrow{t} \leftarrow \ldots \qquad \texttt{let value\_of\_t} = \ldots$$
$$\overleftarrow{t} \leftarrow \ldots \qquad \texttt{let t\_of\_value} = \ldots$$

Type `loc` is left abstract as it is implemented as `Obj.t`: the location of an object is the reference cell where it is stored. Comparing locations has to be done using the OCaml physical equality operator `==`. The function `value_of_t` uses some unsafe features of OCaml to store all the values already seen in an untyped way, when unfolding the value. The `t_of_value` function also uses some unsafe features of OCaml , but only when cyclic ML values are built from a value of type `Value.t`. However, this is hidden in the generated library code and never exposed to the end-user programmer.

4 SQL Persistence

We now describe 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, and the user does not write any 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 ⋆)                                              OCaml
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
```

The `t_init` function connects to the database to check if values of type named `t` have already been persisted; if so, it checks if the *structure* of `t` is consistent with previously persisted values of types named `t`, that is if values of the current type `t` can be safely stored and/or read into the database. It performs this schema check using the structural sub-typing-aware type redirections described earlier (§2.2). If the database is new, it constructs new tables in the database with the right schema.

This automatic translation between ML types to SQL schemas is described in more detail later (§4.1).

The `t_get` function has a part of its signature left unspecified; this is because the type of the query arguments are parameterised 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 in a read-only database.

The `t_save` function stores values into the database; it uses mutability information exposed by the *type-of* library to perform sharing optimisation when possible: for immutable values, our scheme use hash-consing [7,10] to save memory space. Implementing correctly (mutable) value updates has been an interesting challenge. Consider the following piece of code:

```
(* Type defintions *)                                    OCaml
type x = { mutable x : string } with orm
type t = { a : int; b : x } with orm
(* Code *)
let t = { a = 0; b = { x = "foo" } }
and db = t_init "mydb.db" in
t_save db t;
t.b.x ← "bar";
t_save db t
```

This should create only one record of each type in the database; the second call to `t_save` needs to detect that the value of type `x` is at the same location but has different content. Our implementation uses: (*i*) a hidden global cache, associating unique identifiers to ML values for a given database name; (*ii*) weak pointers to clean this cache when a value is garbage-collected; and (*iii*) SQL triggers to update the cache correctly when new values are deleted or added.

The `t_delete` function raises interesting implementation problems as well. In a garbage-collected language, it is not clear how to mix automatic memory management with persistent values. We cannot rely on liveness analysis and life-propagation algorithms to know if an object can be deleted or not, as the purpose of persisting objects is to make the life of a value longer than the program which created it. Conventional counting mechanisms also do not work when cyclic values are present. Our implementation uses a mix of these two techniques, but we view the precise semantics of a “persistent deletion” function as an open challenge, as it can be confusing for the programmer to determine the behaviour without knowing the details of our framework.

4.1 Schema creation

SQL schemas are automatically constructed when connecting to a new database. Let us suppose we have a set of column and table names $\mathcal{N}$ such that $0, 1 \in \mathcal{N}$ and such that they verifies the following property : if $n \in \mathcal{N}$ and $m \in \mathcal{N}$, then

$n \cdot m \in \mathcal{N}$. Then, the schema creation syntax of SQL can be defined as :

$$\texttt{sql} ::= \begin{array}{l} t \vdash (c_1 : \texttt{type}) \cdot \ldots \cdot (c_k : \texttt{type}) \\ \texttt{sql}; \texttt{sql} \end{array} \qquad t \in \mathcal{N}, c_i \in \mathcal{N} \quad \begin{array}{l} \text{table creation} \\ \text{sequence} \end{array}$$

$$\texttt{type} ::= I(i) \mid R \mid T \mid F(t) \mid \bot \qquad i \in \mathbb{N}, t \in \mathcal{N} \quad \text{column type}$$

where $I(i)$ stands for an i-bit integers, R for reals, T for texts, $F(t)$ for row IDs of foreign tables and $\bot$ for binary data. Furthermore, to be valid, a creation query needs to verify that every column of the created table has a unique name:

Validity Property : $t \vdash (c_1 : t_1) \cdot \ldots \cdot (c_k : t_k)$ is *valid* if $i \neq \jmath$ implies that $c_i \neq c_j$

We can now describe how to translate any ML type (with no free type parameters) into a *valid* SQL statement. Figure 1 shows how to inductively build the collection of fields from a name and an element of `tt`, i.e. it defines a function $\mathcal{F} : \mathcal{N} \times \texttt{tt} \to (\mathcal{N} \times \texttt{type})^\star$. Equations (1)-(4) translates basic constructors of `tt` into simple fields with the appropriate type; Equations (5), (8) and (9) 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, equations (6)-(7) fold the induction through the sub-terms of the current term of type `tt`, and propagate the name changes. We ensure the validity property by giving a different field name to each sub-induction call.

$$\mathcal{F}_n(\texttt{INT}(i)) = (n : I(i)) \tag{1}$$
$$\mathcal{F}_n(\texttt{FLOAT}) = (n : R) \tag{2}$$
$$\mathcal{F}_n(\texttt{STRING}) = (n : T) \tag{3}$$
$$\mathcal{F}_n(t_1 \to t_2) = (n : \bot) \tag{4}$$
$$\mathcal{F}_n([t]) = (n \cdot 0 : F(n \cdot 0)) \tag{5}$$
$$\mathcal{F}_n(\langle m_1 : t_1 \rangle \times \ldots \times \langle m_k : t_k \rangle) = \mathcal{F}_{n \cdot m_1}(t_1) \cdot \ldots \cdot \mathcal{F}_{n \cdot m_k}(t_k) \tag{6}$$
$$\mathcal{F}_n(\langle m_1 : t_1 \rangle + \ldots \langle m_k : t_k \rangle) = (n \cdot 0 : T) \cdot \mathcal{F}_{n \cdot m_1}(t_1) \cdot \ldots \cdot \mathcal{F}_{n \cdot m_k}(t_k) \tag{7}$$
$$\mathcal{F}_n(\mathit{type}\ \rho \cdot t) = (n : F(n)) \tag{8}$$
$$\mathcal{F}_n(\rho) = (n : F(n)) \tag{9}$$

Fig. 1. Field semantics for schema creation

Figure 2 shows how to build the set of SQL tables from a name and element of `tt`, i.e. it defines a function $\mathcal{T} : \mathcal{N} \times \texttt{tt} \to \texttt{sql}$. In equations (10) and (11), basic constructors of `tt` do not affect the set of tables (they are handled in the previous field semantics). Equations (12) and (15) create foreign tables t, which are referenced as fields of type $F(t)$ from the field semantics in Equations (5),(8) and (9). Moreover, equation (12) adds the field $(n \cdot 0 : F(n \cdot 0))$ to the field semantics of the $n \cdot 0$ table: an enumeration is stored as a simply linked list in the database,

the *next* relation being stored in that new field. Equations (13) and (14) also fold the induction through the current term as with the semantics in (6) and (7).

$$\mathcal{T}_n(\texttt{base}) = \emptyset \tag{10}$$
$$\mathcal{T}_n(t_1 \to t_2) = \emptyset \tag{11}$$
$$\mathcal{T}_n([t]) = n \cdot 0 \vdash (n \cdot 0 : F(n \cdot 0)) \cdot \mathcal{F}_{n \cdot 1}(t) \ ; \ \mathcal{T}_{n \cdot 0}(t) \tag{12}$$
$$\mathcal{T}_n(\langle m_1 : t_1 \rangle \times \ldots \times \langle m_k : t_k \rangle) = \mathcal{T}_{n \cdot m_1}(t_1) \ ; \ \cdots \ ; \ \mathcal{T}_{n \cdot m_k}(t_k) \tag{13}$$
$$\mathcal{T}_n(\langle m_1 : t_1 \rangle + \ldots + \langle m_k : t_k \rangle) = \mathcal{T}_{n \cdot m_1}(t_1) \ ; \ \cdots \ ; \ \mathcal{T}_{n \cdot m_k}(t_k) \tag{14}$$
$$\mathcal{T}_n(\mathit{type}\ \rho \cdot t) = n \vdash \mathcal{F}_n(t) \ ; \ \mathcal{T}_n(t) \tag{15}$$
$$\mathcal{T}_n(\rho) = \emptyset \tag{16}$$

Fig. 2. Table semantics for SQL

Finally, the SQL queries which create the tables to persist values of type ρ is $\mathcal{T}_\rho(\underline{\rho})$, where $\underline{\rho}$ is the dynamic type as computed in §2.2.

5 Example: Photo Gallery

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:

```sql
CREATE TABLE image (__id__ INTEGER PRIMARY KEY, image TEXT);
CREATE TABLE gallery (__id__ INTEGER PRIMARY KEY,
    gallery__name TEXT, gallery__date REAL, gallery__contents__0 INTEGER);
CREATE TABLE gallery__contents__0 (__id__ INTEGER PRIMARY KEY,
    __next__ INTEGER, __size__ INTEGER, gallery__contents__0 INTEGER);
```

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
```

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
```

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

The above code snippet saves a gallery named “Leonardo” containing an unique fresh image in a database called `louvre.db`. It then queries all the galleries whose name is strictly equal to “Leonardo”. 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 [20] and MetaOCaml [21]. MetaHDBC [14] 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* [24] which is inspired by the construct in the same name in Haskell [11]. Language-integrated constructs to manipulate databases is also an active topics for mainstream languages, such as the LINQ [16] 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] M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *J. Funct. Program.*, 5(1):111–130, 1995.

[2] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *LICS*, pages 242–252, 1996.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[4] S. Blott. Type inference and type classes. In K. Davis and J. Hughes, editors, *Functional Programming*, Workshops in Computing, pages 254–265. Springer, 1989.

[5] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[7] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

[8] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

[9] T. Gazagnaire and V. Hanquez. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 203–214. ACM, 2009.

[10] E. Goto. Monocopy and associative algorithms in an extended LISP. Technical Report 74-03, Information Sciences Lab., University of Tokyo, 1974.

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

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

[13] X. Leroy and M. Mauny. DynamicsOA in ML. *J. Funct. Program.*, 3(4):431–463, 1993.

[14] M. Lindstrom. MetaHDBC: Statically Checked SQL for Haskell (Draft). 2008.

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

[16] 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.

[17] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

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

[19] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions. In *The 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

[20] 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.

[21] 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.

[22] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[23] G. Winskel. *The formal semantics of programming languages.* The MIT Press, 1993.

[24] J. Yallop. Practical generic programming in OCaml. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 83–94. ACM, 2007.