

# Runtime Types in OCaml

Alain Frisch (LexiFi)

Meeting of the Caml Consortium – November 2011

## Overview

Runtime representation of types

Building and passing runtime representation of types

Application: Dynamics, variants

Abstract types and type variables

Conclusion and open questions

## Types at runtime?

- ▶ The OCaml execution model is untyped.
- ▶ I'm **not** proposing to attach type information to values.
- ▶ Proposal:
  - ▶ Define a datatype to represent the structure of static types at runtime.
  - ▶ Give easy ways to obtain the representations of types and to pass them around.
  - ▶ Provide functions to manipulate runtime types in a type-safe way.

## Examples of applications

- ▶ Generic “print” function.
- ▶ Type-safe deserialization (cf work by Grégoire Henry).
- ▶ Generic comparison function with custom behavior for specific types.
- ▶ Universal tagged representation of values (variants).
- ▶ Automatic generation of GUIs or SQL schema / queries.

## Runtime Types at LexiFi

- ▶ A system close to this proposal has been in use at LexiFi for several years and is now quite stable.
- ▶ We are not proposing a direct inclusion. Our experience gave us insights on what should be done better.
- ▶ Most of our interesting general-purpose libraries developed internally (and which we might be willing to open-source) rely on runtime types in some ways.
- ▶ The patch to the compiler is very small. No invasive change to the compilation strategy.

## Overview

### Runtime representation of types

Building and passing runtime representation of types

Application: Dynamics, variants

Abstract types and type variables

Conclusion and open questions

## Building runtime representation of types

- ▶ The standard library exposes an abstract datatype `'a ty`.
- ▶ A value of type `t ty` represents the structure of the type `t`.
- ▶ Functions to operate on `'a ty`: equality test, composition, decomposition.

```
1 val int_ty: int ty
2 val string_ty: string ty
3 val list_ty: 'a ty -> 'a list ty
4 ...
```

## Dynamic type equality

```
1 (* In stdlib: *)
2 type (_, _) eq = Eq: ('a, 'a) eq (* witness of equality *)
3 val types_equal: 'a ty -> 'b ty -> ('a, 'b) eq option
4
5 (* Example of a generic function: *)
6 let print (type t) (t : t ty) (x : t) =
7   match types_equal t int_ty with
8   | Some Eq -> print_int x
9   | None ->
10     match types_equal t string_ty with
11     | Some Eq -> print_string x
12     | None -> print_string "<other>"
```

## Decomposing type constructors

```
1 type _ is_list = Is_list: 'a ty -> 'a list is_list
2 val is_list: 'a ty -> 'a is_list option
3
4 let rec print: 'a. 'a ty -> 'a -> unit =
5   fun (type t) (t : t ty) (x : t) ->
6     ...
7     match is_list t with
8     | Some (Is_list s) -> List.iter (print s) x
9     | None -> ...
```

## Decomposing record types

```
1 type _ is_record = Is_record: 'a field list -> 'a is_record
2 and 'a field = Field: ('a, 'b) field_ -> 'a field
3 and ('a, 'b) field_ = {
4     label: string;
5     field_type: 'b ty;
6     get: ('a -> 'b);
7 }
8 val is_record: 'a ty -> 'a is_record option
9
10 let rec print: 'a. 'a ty -> 'a -> unit =
11     ...
12     match is_record t with
13     | Some (Is_record fields) ->
14         List.iter
15             (fun (Field {label; field_type; get; _}) ->
16                 Printf.printf " %s=" label;
17                 print field_type (get x)
18             )
19         fields
20     | None -> ...
```

## Decomposing record types

Exercise for the reader:

- ▶ Extend the representation of record types to support creating value (resp. modifying mutable fields).
- ▶ Support for tuple (arbitrary length) and sum types.

## All together

```
1 type 'a head =  
2   | Int: int head  
3   | String: string head  
4   | List: 'a ty -> 'a list head  
5   | Option: 'a ty -> 'a option head  
6   | Tuple2: ('a ty * 'b ty) -> ('a * 'b) head  
7   | Tuple3: ('a ty * 'b ty * 'c ty) -> ('a * 'b * 'c) head  
8   | Record: 'a record -> 'a head  
9   | ...  
10  
11 val head: 'a ty -> 'a head
```

## Untyped representation

```
1 type uty =  
2   | Int  
3   | String  
4   | List of uty  
5   | Option of uty  
6   | Tuple of uty list  
7   | Record of (string * bool * uty) list  
8   | ...  
9  
10 val untyped: 'a ty -> uty
```

Can be useful e.g. to generate source code that traverse types, for low-level algorithms (using Obj module), or for creating hash-tables indexed by types.

## Custom behavior for generic functions

```
1 type custom_printer = Custom_printer: 'a ty -> ('a -> unit)  
  -> custom_printer  
2  
3 val print: 'a ty -> ?custom_printers:custom_printer list -> '  
  a -> unit  
4   (* Explicit list of custom printers. *)  
5  
6 val register_custom_printer: 'a ty -> ('a -> unit) -> unit  
7   (* Global registration. *)
```

Overview

Runtime representation of types

Building and passing runtime representation of types

Application: Dynamics, variants

Abstract types and type variables

Conclusion and open questions

## Manually building representation?

```
1 val int_ty: int ty
2 val string_ty: string ty
3 val list_ty: 'a ty -> 'a list ty
4 ...
5 val of_head: 'a head -> 'a ty
```

- ▶ Tedious.
- ▶ For constructed typed, very tedious + we don't want the internal representation to contain closures.
- ▶ Representing recursive types (in a way that allows to manipulate them as finite structures) is difficult.

## Building type representation automatically

- ▶ The programmer can write the expression `(type of t)` to build the representation of type  $t$  (a type expression).
- ▶ The type  $t$  must not contain type variables (after unification).

```
1 let () = print (type of _) [1; 2; 3]
2   (* _ is unified to int list *)
```

## Automatic type representation arguments

```
1 val print: #t:'a ty -> 'a -> unit
2
3 let () = print [1; 2; 3]
```

- ▶ #-arguments are 'automatic' arguments. When not provided explicitly on the call site, they are synthesized by the compiler (here by inserting  $\sim t : (\text{type of } \_)$ ).
- ▶ In this proposal, automatic arguments are only allowed to have an  $\_ \text{ ty}$  type. Other uses are possible (e.g. a type of automatic argument to insert a description of the call site location).
- ▶ Alternative: a labelled argument is automatically automatic if it has type  $\_ \text{ ty}$  (need to change the semantics of labelled non-optional arguments to avoid confusion with partial application).

## Overview

Runtime representation of types

Building and passing runtime representation of types

**Application: Dynamics, variants**

Abstract types and type variables

Conclusion and open questions

## Dynamics

- ▶ Pairing a value and the runtime representation of its type.
- ▶ Useful for generic type-driven unary traversal of values.

```
1 type dyn = Dyn: 'a ty * 'a -> dyn
2 val dyn: #t:'a ty -> 'a -> dyn
3
4 let print_dyn (Dyn (ty, v)) = print ty v
5
6 val cast: #t:'a ty -> dyn -> 'a option
7 let cast (type t) #t:(exp_ty : t ty) (Dyn (ty, v)) =
8   match types_equal exp_ty ty with
9   | Some Eq -> Some v
10  | None -> None
11
12 type dyn_head =
13   | Int of int
14   | Tuple of dyn
15   | List of dyn
16   | ...
17 val dyn_head: dyn -> dyn_head
```

# Variants

- A universal tagged representation of values.

```
1 type variant =  
2   | Int of int  
3   | String of string  
4   | List of variant list  
5   | Tuple of variant list  
6   | Record of (string * variant) list  
7   | ...  
8  
9 val to_variant: #t:'a ty -> 'a -> variant  
10 val of_variant: #t:'a ty -> variant -> 'a
```

## Overview

Runtime representation of types

Building and passing runtime representation of types

Application: Dynamics, variants

**Abstract types and type variables**

Conclusion and open questions

## Abstract types with transparent representation

- ▶ A module which exposes an abstract type can also expose its representation.
- ▶ One needs a way to allow the compiler to use this representation when building larger type representations.

```
1 module X : sig type t   val t_repr: t ty end = ...  
2  
3 let x = (type of (string * X.t) list with X.t_repr)  
4 let x = let type X.t_repr in (type of (string * X.t) list)
```

- ▶ (Extra proposal? When the compiler needs to create the representation of an abstract type `X.t`, it looks for a value with the same path and of type `X.t ty`.)

## Type variables

- ▶ One can use the same mechanism with type variables (reified as local abstract types).

```
1  let mk_list (type a) (a_repr : a ty) =  
2    (type of a list with a_repr)
```

- ▶ (In LexiFi's version, one can also plug runtime representation for type variables directly, without reifying them as local abstract types, but this is fragile.)

## Abstract types with opaque representation

- ▶ A way to build a fresh representation for an abstract type (different from all other representations), local to the current process. The programmer is in charge of managing the “generativity”.

```
1 val new_opaque: unit -> 'a ty
```

- ▶ A module could expose an abstract type together with an “opaque” representation for it.

## Abstract types with opaque representation

```
1 val register_custom_printer: 'a ty -> ('a -> unit) -> unit
2
3 module X : sig
4   type t
5   val t_repr: t ty
6   ...
7 end = struct
8   type t = {x: int; y: int}
9   let t_repr : t ty = new_opaque ()
10  let () = register_custom_printer t_repr (fun r -> print r)
11 end
```

## Opaque type variables

```
1 let show_assoc_list (type a) f (l : (string * a) list) =  
2   let a_repr : a ty = new_opaque () in  
3   let type a_repr in  
4   print ~custom_printers:[Custom_printer (a_repr, f)] l  
5  
6 val show_assoc_list: ('a -> string) -> (string * 'a) list ->  
   unit
```

## Semi-opaque abstract types

```
1 val new_opaque: ?repr:'a ty -> unit -> 'a ty
```

- ▶ The optional `repr` argument could be used to allow built-in low-level operations (like type-safe unmarshalling) or explicitly revealing structure (for debugging, etc). An alternative would be to register the structure in a global store (or to pass it explicitly where needed).

## Parametrized abstract types

```
1 module type OPAQUE1 = sig
2   type 'a constr
3   type _ is_constr = Is_constr: 'a ty -> 'a constr is_constr
4   val make: 'a ty -> 'a constr ty
5   val is_constr: 'a ty -> 'a is_constr option
6 end
7
8 module Opaque1(X : sig type 'a constr end) : OPAQUE1
```

## Customizing generic functions

```
1 module type CustomPrinter1 = sig
2   module O: OPAQUE1
3   val print: 'a ty -> 'a O.constr -> unit
4 end
5
6 type custom_printer =
7   | Custom_printer: 'a ty * ('a -> unit) -> custom_printer
8   | Custom_printer1: (module CustomPrinter1) ->
      custom_printer
```

## Overview

Runtime representation of types

Building and passing runtime representation of types

Application: Dynamics, variants

Abstract types and type variables

Conclusion and open questions

## Changes to the compiler

- ▶ Small changes to the compiler:
  - ▶ (type of ... with ...)
  - ▶ let type ... in ... (optional).
  - ▶ Automatic type representation arguments (optional).
- ▶ Plus a new module in the stdlib.

## Concrete internal representation

- ▶ Representation of datatypes, recursive types.
- ▶ Exotic types (objects, 1st class modules, polymorphic variants, GADTs, records with polymorphic fields).
- ▶ Keeping a notion of type name (path?) for datatypes.
- ▶ Dealing with private datatypes.
- ▶ Unique ids in the internal representation for efficient type-indexed tables.
- ▶ Hash-consing, maximal sharing of the internal representation.

## Meta-data

- It is convenient to customize the behavior of generic functions with annotations put on type declarations. This could share the same syntax as a generic extension of OCaml AST with meta-data.

```
1 val head_meta_data: 'a ty -> (string * string) list
2 val add_head_meta_data: (string * string) list -> 'a ty -> 'a
  ty
3
4 type t = {
5   x @(gui_name="X-axis"): int;
6   y @(gui_name="Y-axis"): int;
7 }
```

## Type paths

```
1 type ('a, 'b) path
2 val follow_value: 'a -> ('a, 'b) path -> 'b
3 val follow_type: 'a ty -> ('a, 'b) path -> 'b ty
4
5 type t = {x: int; y: s}
6 and s = {a: int; b: int list}
7
8 let p : (t, int) path = (.y.a)
9 let p : (t, int) path = (.y.b[0])
```

- ▶ Values that describes navigation paths in types/values.
- ▶ Applications: access sub-components of generated GUI; describe SQL queries; etc.

## Other questions

- ▶ Explicit syntax for automatic type arguments vs. type-driven?
- ▶ `let type e in ...`, or (only) explicit `(type of t with e)`?