Runtime Types in OCaml

Alain Frisch (LexiFi)

Meeting of the Caml Consortium – November 2011

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.

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 of `'a ty`: equality test, composition, decomposition.

```
val int_ty: int ty
val string_ty: string ty
val list_ty: 'a ty -> 'a list ty
...
```

Dynamic type equality

```
(* In stdlib: *)
type (_, _) eq = Eq: ('a, 'a) eq  (* witness of equality *)
val types_equal: 'a ty -> 'b ty -> ('a, 'b) eq option

(* Example of a generic function: *)
let print (type t) (t : t ty) (x : t) =
  match types_equal t int_ty with
  | Some Eq -> print_int x
  | None ->
      match types_equal t string_ty with
      | Some Eq -> print_string x
      | None -> print_string "<other>"
```

Decomposing type constructors

```
type _ is_list = Is_list: 'a ty -> 'a list is_list
val is_list: 'a ty -> 'a is_list option

let rec print: 'a. 'a ty -> 'a -> unit =
  fun (type t) (t : t ty) (x : t) ->
  ...
        match is_list t with
        | Some (Is_list s) -> List.iter (print s) x
        | None -> ...
```

Decomposing record types

```
type _ is_record = Is_record: 'a field list -> 'a is_record
and 'a field = Field: ('a, 'b) field_ -> 'a field
and ('a, 'b) field_ = {
    label: string;
    field_type: 'b ty;
    get: ('a -> 'b);
   }
val is_record: 'a ty -> 'a is_record option

let rec print: 'a. 'a ty -> 'a -> unit =
  ...
        match is_record t with
        | Some (Is_record fields) ->
            List.iter
              (fun (Field {label; field_type; get; _}) ->
                Printf.printf " %s=" label;
                print field_type (get x)
              )
              fields
        | None -> ...
```

Decomposing record types

Exercise for the reader:

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

All together

```
type 'a head =
  | Int: int head
  | String: string head
  | List: 'a ty -> 'a list head
  | Option: 'a ty -> 'a option head
  | Tuple2: ('a ty * 'b ty) -> ('a * 'b) head
  | Tuple3: ('a ty * 'b ty * 'c ty) -> ('a * 'b * 'c) head
  | Record: 'a record -> 'a head
  | ...

val head: 'a ty -> 'a head
```

Untyped representation

```
type uty =
  | Int
  | String
  | List of uty
  | Option of uty
  | Tuple of uty list
  | Record of (string * bool * uty) list
  | ...

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

```ocaml
type custom_printer = Custom_printer: 'a ty -> ('a -> unit) -> custom_printer

val print: 'a ty -> ?custom_printers:custom_printer list -> 'a -> unit
  (* Explicit list of custom printers. *)

val register_custom_printer: 'a ty -> ('a -> unit) -> unit
  (* Global registration. *)
```

Manually building representation?

```
val int_ty: int ty
val string_ty: string ty
val list_ty: 'a ty -> 'a list ty
...
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).

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

Automatic type representation arguments

```
val print: #t:'a ty -> 'a -> unit

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 `~t:(type of _)`).
- In this proposal, automatic arguments are only allowed to have an `_ 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 `_ 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.

```
type dyn = Dyn: 'a ty * 'a -> dyn
val dyn: #t:'a ty -> 'a -> dyn

let print_dyn (Dyn (ty, v)) = print ty v

val cast: #t:'a ty -> dyn -> 'a option
let cast (type t) #t:(exp_ty : t ty) (Dyn (ty, v)) =
  match types_equal exp_ty ty with
  | Some Eq -> Some v
  | None -> None

type dyn_head =
    | Int of int
    | Tuple of dyn
    | List of dyn
    | ...
val dyn_head: dyn -> dyn_head
```

Variants

- A universal tagged representation of values.

```
type variant =
  | Int of int
  | String of string
  | List of variant list
  | Tuple of variant list
  | Record of (string * variant) list
  | ...

val to_variant: #t:'a ty -> 'a -> variant
val of_variant: #t:'a ty -> variant -> 'a
```

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 representation.

```
module X : sig  type t_repr   val t: t ty  end = ...

let x = (type of (string * X.t) list with X.t_repr)
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 (reifed as local abstract types).

```
let mk_list (type a) (a_repr : a ty) =
  (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”.

```
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

```
val register_custom_printer: 'a ty -> ('a -> unit) -> unit

module X : sig
  type t
  val t_repr: t ty
  ...
end = struct
  type t = {x: int; y: int}
  let t_repr : t ty = new_opaque ()
  let () = register_custom_printer t_repr (fun r -> print r)
end
```

Opaque type variables

```
let show_assoc_list (type a) f (l : (string * a) list) =
  let a_repr : a ty = new_opaque () in
  let type a_repr in
  print ~custom_printers:[Custom_printer (a_repr, f)] l

val show_assoc_list: ('a -> string) -> (string * 'a) list ->
    unit
```

Semi-opaque abstract types

```
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

```
module type OPAQUE1 = sig
  type 'a constr
  type _ is_constr = Is_constr: 'a ty -> 'a constr is_constr
  val make: 'a ty -> 'a constr ty
  val is_constr: 'a ty -> 'a is_constr option
end

module Opaque1(X : sig type 'a constr end) : OPAQUE1
```

Customizing generic functions

```
module type CustomPrinter1 = sig
  module O: OPAQUE1
  val print: 'a ty -> 'a O.constr -> unit
end

type custom_printer =
  | Custom_printer: 'a ty * ('a -> unit) -> custom_printer
  | Custom_printer1: (module CustomPrinter1) ->
      custom_printer
```

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.

Other questions

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

```
val head_meta_data: 'a ty -> (string * string) list

type t = {
  x @(gui_name="X-axis"): int;
  y @(gui_name="Y-axis"): int;
}
```