Runtime types in OCaml.

Grégoire Henry (Inria)
Jacques Garrigue (Nagoya University)

with support from LexiFi and OCamlPro

OCaml’2013 – September 24 – Boston
Dynamic typing in statically typed languages: what for?

**Structural type introspection** e.g. generic Input/Output primitives:
- type-safe, unlike Marshal.{to,from}_string
- typed, i.e. not written in Camlp4
  a.k.a. polytypic functions

**Nominal type introspection** e.g. dynamic values:
- dynamic key/value storage
- to implement a DSL with dynamic typing
  or extensible polytypic functions (for abstract type)

**A common type representation** for:
- FFI libraries
- Eliom’s services

**Debugger** explore the heap with (exact) typing information
Main problem

Is there a single representation that fits all these usages?

- while preserving abstraction when wished
- while breaking abstraction when wished (but not by mistake)
- without hidden cost
let rec print (type t) (ty: t ty) (v: t) =
match head ty with
| Int → print_int v
| String → print_string v
| List ty → print_list (print ty) v
| ...

Small example: polytypic printing function
let rec print (type t) (ty: t ty) (v: t) =
    match head ty with
    | Int  → print_int v
    | String  → print_string v
    | List ty → print_list (print ty) v
    | Sum desc →
        let (name, args) = sum_get desc v in
        print_string name;
        if List.length args <> 0 then
            printf "(%a)" print_args args
    | ...

and print_args = function
    | [Dyn (ty, v)] → print ty v
    | Dyn (ty, v) :: args →
        print ty v; printf ","; print_args args
    | [] → assert false
let rec print (type t) (ty: t ty) (v: t) =
  match head ty with
  | Int → print_int v
  | String → print_string v
  | List ty → print_list (print ty) v
  | Sum desc →
    let (name, args) = sum_get desc v in
    print_string name;
    if List.length args <> 0 then
      printf "(\%a)" print_args args
  | ...
  | Abstract → print_string "<abstract>"
and print_args = function
  | [Dyn (ty, v)] → print ty v
  | Dyn (ty, v) :: args →
    print ty v; printf ","; print_args args
  | [] → assert false
A type for types: the predefined type \( \tau \) ty;

An syntax for type expression: \((\text{type val } \tau)\) of type \( \tau \) ty, the runtime representation of \( \tau \).

type t = R of int * int | ...
let x = R (22, 7)
let () = print (type val t) x
Implicit type arguments: optional argument instantiated at call-site with the dynamic representation of the expected type.

```ocaml
val print: ?t:(type val α) → α → string

let print ?(type val t) (v: t) = ...

type t = R of int * int | ...
let x = R (22, 7)
let () = print x (* implicit arg is (type val t) *)
```
Main problem, reformulated.

How to mix polyporphic function and abstraction?

- without always printing <abstract>
- and given that a data type may have multiple and distinct abstract representation

```latex
module type INTF = sig type t ... end
module IMPLEM = struct type t = ... end
module M = (IMPLEM : INTF)
module M2 = (IMPLEM : INTF)
```
module M : sig
  type t
  val x : t
end = struct
  type t = R of int * int | ...
  let x = R (22, 7)
  let () = print x  (* display: “R (22, 7)” *)
end
let () = print x  (* display: “<opaque>” *)
module M : sig
  type t
  val x : t
end = struct
  type t = R of int * int | ...
  let x = R (22, 7)
  let () = print x (* display: “R (22, 7)” *)
  let () =
    register_printer (external type val t)
    (fun x → ...)
end
  let () = print x (* display: “R (22, 7)” *)

May be implemented with type-indexed association table.
Abstract type and nominal type introspection (1/3)

Which relation between a data type and its abstraction(s)?

```ocaml
module type INTF = sig
  type t
  val x : t
end

module IMPLEM = struct
  type t = R of int * int | ...
  let x = R (22, 7)
end

include (IMPLEM : INTF)
```
Abstract type and nominal type introspection (1/3)

Which relation between a data type and its abstraction(s)?

```ocaml
module type INTF = sig
  type t
  val x : t
end

module IMPLEM = struct
  type t = R of int * int | ...
  let x = R (22, 7)
end

include (IMPLEM : INTF)

let cast ?(type val a) (x : a) : IMPLEM.t option =
  match (type val a) with
  | (type val IMPLEM.t) → Some (x : IMPLEM.t)
  | _ → None
```
Which relation between a data type and its abstraction(s) ?

module type INTF = sig
  type t
  val x : t
end

module IMPLEM = struct
  type t = R of int * int | ...
  let x = R (22, 7)
  let is_t ?(type val a) (x : a) =
    match (type val a) with
    | (type val t) → true
    | _ → false
end

include (IMPLEM : INTF)
Alias type have no proper identity:

module M : sig
  type t
  val x : t
end = struct
  type t = int list
  let x = [1;2;3]
end

let cast ?(type val a) (x : a) : int list option =
  match (type val a) with
  | (type val int list) → Some (x : int list)
  | _ → None
Global context  There is a canonical name for type defined outside of the current compilation unit: its absolute path.

Wish  By default abstraction should consistently introduces new nominal types. But, how to reference (all) the external name(s) of a given type within its initial compilation unit/structure?
Abstract type and nominal type introspection: summary.

Global context  There is a canonical name for type defined outside of the current compilation unit: its absolute path.

Wish  By default abstraction should consistently introduces new nominal types. But, how to reference (all) the external name(s) of a given type within its initial compilation unit/structure?

Pragmatic approach  manual or semi-automatic creation of runtime “type names”

▶ track nominal usage of type
▶ annotate signature accordingly
Behind the scene

An unsafe type for type

```ocaml
type uty =
    | DT_Bool | DT_Int | DT_List of uty
    ...
    | DT_Constr of declaration * uty list
    | DT_Var of var_id
and declaration =
    { decl_id = id;
      params = var_id list;
      kind = kind; }
and kind = DT_Sum of ... | DT_Record of ...
```

Absolute path as type identifiers

```ocaml```

and id = string list * string
Conclusion: what’s working?

- Runtime type representation with global names
- A GADT for structural introspection
- Type-constructor indexed association table
- Implicit type argument
  - lightweight syntax for calling polytypic function
  - explicit type parameter for polymorphic function