Dynamic typing in OCaml.

Grégoire Henry, Jacques Garrigue.

名古屋大学

2013, May 7.
Dynamic typing in statically typed language: what for?

- type-safe input/output primitives
- dynamic values:
  - for implementing a DSL with dynamic typing
  - dynamic key/value storage
Roadmap

**Basic operations**: how to manipulate types in a type-safe manner?
- testing type equality
- nominal type introspection
- structural type introspection

**Behind the scene**: how to implement these primitives?

**Extensibility**: type-indexed association table.

**Identity**: how to define type equality in the presence of:
- type aliases?
- abstracted types?
- functor?
- first-class module?
Basic ingredients for dynamic typing in statically type languages.

We extend the language with:

A type for types: the predefined singleton type $\tau$ $ty$;

An expression: $(\text{type } \tau)$ of type $\tau$ $ty$. Its evaluation is the runtime representation of the type $\tau$.

A equality predicate: $\text{Ty.eq}$ of type $\alpha$ $ty \rightarrow \beta$ $ty \rightarrow$ bool.
Exemple : marshaling dynamic values.

Given the following unsafe (un)marshaling primitives:

```plaintext
val unsafe_marshal : α → string
val unsafe_unmarshal : string → α
```

One may implement safestest (un)marshaling functions:

```plaintext
let marshal : α ty → α → string = fun ty v → unsafe_marshal (ty, v)
let unmarshal : α ty → string → α = fun ty s →
  let (ty’, v) : β ty * β = unsafe_unmarshal s in
  if not (Ty.eq ty ty’) then failwith "Type error";
  v

let s = marshal (type int list) [1;2;3]
let l1 = unmarshal (type int list) s
let l2 = unmarshal (type bool list) s
```
Adding salt: implicit type arguments.

To ease the usage of dynamically typed functions, we may also extend the language with:

**Implicit type arguments**: optional argument implicitly instantiated at call-site with the dynamic representation of the expected type.

```ocaml
define marshal: (?\(\alpha\)) \to \alpha \to string
define unmarshal: (?\(\alpha\)) \to string \to \alpha

let marshal ?(type t) (v: t) =
    unsafe_marshal ((type t), v)
let unmarshal ?(type t) (s: string) : t = ...

let s = marshal [1;2;3]
let l1 = 1 :: unmarshal s
let l2 = true :: unmarshal s
```
By default, polymorphic functions do not have access to the dynamic representation of type variable. Hence, the following snippet do not compile:

```ocaml
let wrong_apply (f: α list → β) (x: string) : β =
    let l = unmarshal x in (* implicit = (α list) *)
    f (l @ l)
```

The dynamic type parameter should be explicit:

```ocaml
let apply ?(type t) (f: t list → β) (x: string) : β =
    let l = unmarshal x in (* implicit = (t list) *)
    f (l @ l) let sum_list = List.fold_left (+) 0
    let twelve = apply sum_list (marshal [1;2;3])
```
let print ?(type t) (v : t) = 
  if Ty.eq (type t) (type int) then 
    print_int v 
  else if Ty.eq (type t) (type float) then 
    print_float v 
  else if Ty.eq (type t) ... then 
    ...
  else print_string "<abstract>"
Nominal type introspection: first attempt.

```
let print ?(type t) (v : t) =
  if Ty.eq (type t) (type int) then
    print_int v
  else if Ty.eq (type t) (type float) then
    print_float v
  else if Ty.eq (type t) ... then
    ...
  else print_string "<abstract>"
```

This function is not well typed.
Nominal type introspection : type-equality witness.

We have to refine the type of the type-equality predicate:

\[
\text{type } (_ , _) \text{ eq } = \text{Eq: } (\alpha, \alpha) \text{ eq}
\]
\[
\text{val eq: } \alpha \text{ ty } \to \beta \text{ ty } \to (\alpha, \beta) \text{ eq option}
\]

GADT as a type-equality witness allows to write well-typed generic functions:

\[
\text{let print?(type t) (v : t) =}
\]
\[
\begin{align*}
\text{match Ty.eq (type t) (type int) with} \\
| \text{Some Eq } & \rightarrow \text{print_int v} \\
| \text{None } & \rightarrow \\
\text{match Ty.eq (type t) (type float) with} \\
| \text{Some Eq } & \rightarrow \text{print_float v} \\
| \text{None } & \rightarrow \ldots
\end{align*}
\]
We also provide:

a set of instanciation predicate: for each parametrised predefined type \( \_ \_ t \), a function is\_instance allows to test if a type is a type is one of its instance;

a instanciation-predicate builder: a mechanism to build such functions instanciation predicate for user-defined types constructors.
Nominal type introspection: parametrised types (2/2).

Given a module with the following signature:

```ocaml
module List = sig
  type _ inst = Inst: α ty → α list inst
  val is_instance: α ty → α inst option
end
```

We may implement a generic printer that handle all type of list:

```ocaml
let print_list (print_item: α → unit) (l: α list) = ...

let rec print (type t) (ty: t ty) (v : t) =
  match Ty.List.is_inst ty with
  | Some (List.Inst ty’) → print_list (print ty’) v
  | None → ...
```
Dynamic type

Without further extension, one may define dynamic value:

```ocaml
type dyn = Dyn : α ty * α → dyn
let plus : dyn → dyn → dyn =
  fun (Dyn (ty1, v1)) (Dyn (ty2, v2)) →
  match Ty.eq ty1 (type int) with
  | Some Eq →
    begin match Ty.eq ty2 (type int) with
      | Some Eq → Dyn ((type int), v1 + v2)
      | None →
        begin match Ty.eq ty2 (type float) with
          | Some Eq →
            Dyn ((type float), float_of_int v1 +. v2)
          | None → ...
    end
  | None → ...
```

However, it should be lightest with an *ad hoc* syntax.
Structural type introspection: predefined types.

type _ head =
  | Int: int ty
  | String: string ty
  | Array: α ty → α array ty
  | Arrow: α ty * β ty → (α → β) ty
  | ...
  | Record: α record → α ty
  | Sum: α sum → α ty
val head: α ty → α head

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
  | Arrow (ty1, ty2) → print_string "<fun>>"
type \( \alpha \) record = private {
   record_fields: (string * \( \alpha \) field) list;
}

and _ field = private
| Field: \( \beta \) ty * (\( \alpha \), \( \beta \)) field_access \( \to \) \( \alpha \) field
and (\( \alpha \), \( \beta \)) field_access

val field_get: (\( \alpha \), \( \beta \)) field_access \( \to \) \( \alpha \) \( \to \) \( \beta \)
val field_set: (\( \alpha \), \( \beta \)) field_access \( \to \) \( \alpha \) \( \to \) \( \beta \) \( \to \) unit
val field_mutable: (\( \alpha \), \( \beta \)) field_access \( \to \) bool

type field_builder =
   { field_builder: \( \forall \alpha \). \( \exists \) (\( \alpha \)) \( \to \) string \( \to \) int \( \to \) \( \alpha \) }
val record_builder: \( \alpha \) record \( \to \) field_builder \( \to \) \( \alpha \)
let rec print (type t) (ty: t ty) (v: t) =
    match head ty with
    | ...
    | Record r →
        printf "{";
        List.iter (print_record_field v) r.record_fields;
        printf "}";
and print_record_field v (name, Field (ty, access)) =
    printf "%s = " name;
    print ty (field_get access v);
    printf ";"
Structural type introspection: sum (1/2).

type \( \alpha \) sum

type case_arg = CaseArg: \( \alpha \) ty * \( \alpha \) \( \rightarrow \) case_arg
val sum_get: \( \alpha \) sum \( \rightarrow \) \( \alpha \) \( \rightarrow \) string * case_arg list

type _ sum_case = private
  | SumCase_constant: (\( \alpha \), unit) case_sel \( \rightarrow \) \( \alpha \) sum_case
  | SumCase_allocated:
    (\( \alpha \), \( \beta \)) case_sel * \( \beta \) tuple \( \rightarrow \) \( \alpha \) sum_case
and (\( \_ \), \( \_ \)) case_sel
val sum_desc: \( \alpha \) sum \( \rightarrow \) (string * \( \alpha \) sum_case) list

val case_builder: (\( \alpha \), \( \beta \)) case_sel \( \rightarrow \) \( \beta \) \( \rightarrow \) \( \alpha \)
let rec print (type t) (ty: t ty) (v: t) =
  match ty with
  | ...
  | Sum sum →
    let (name, args) = sum_get sum v in
    print_string name;
    if List.length args <> 0 then
      Format.fprintf ppf "(%a)" print_args args
    and print_args args =
      match args with
      | [] → assert false
      | [CaseArg (ty, v)] →
        Format.fprintf ppf "%a" (print ty) v;
      | CaseArg (ty, v) :: args →
        Format.fprintf ppf "%a, %a" (print ty) v
        print_case_args args
  print_case_args args
An unsafe type for type

```plaintext
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

```plaintext
and id = string list * string
```
Considering the following program:

```ocaml
module M = struct
  type q = R of int * int | F of float
  let q_repr = (type q)
end
let q_repr2 = (type M.q)
```

The values `q_repr` and `q_repr2` represents the same type. Their internal representation is:

```
DT_Constr (q_decl, [])
```

where `q_decl` is:

```ocaml
{ decl_id = (["M"],"q");
  params = [];
  kind = DT_Sum ... ; }
```
Extensibility: type-indexed association table (1/3)

We add in the standard library:

A type-indexed association table a specialised Map where key are type expression or type constructor.

Its signature looks like:

```ml
module Make(T : sig type α elt end) = sig
    type t (* Type of the association table. *)
    val find : t → α ty → α T.elt (* Lookup *)
    val add : α ty → α T.elt → t → t (* Insertion *)
    (* ... *)
end
```
Extensibility : type-indexed association table (2/3)

module Make(T : sig type α elt end) = sig

(* ... *)

(* Insertion for type constructor of arity 1. *) module type Constr1 : sig
    type α constr
    val action : ?(α) → α constr T.elt
end
val add1 : (module Constr1) → t → t

(* ... *)

end
module Printers =
   Ty.Make(struct type α elt = α → unit end)
let printers = Printers.create ()

let rec print ?(type t) (v: t) =
   try Printers.find printers (type t) v
   with Not_found → print_string "<abstract>"

let () =
   Printers.add (type int) print_int;
   Printers.add (type float) print_float;
   Printers.add1 (module struct
      type α constr = α list
      let action ?(type t) (l: t list) =
         print list print l
   end)
We described 4 different generic input/output functions based on 4 primitives operations:

- type equalities;
- nominal type introspection;
- structural type introspection;
- type-indexed association table.

Allowing two main usages:

**Dynamic value** as a pair composed of a value and its type.

**Extensible polytypic function** defined by case on the structure of type, that may be extended/specialised by name.
Dynamic type equality and type aliases

Aliases have no dynamic identities The static unification algorithm do not distinguish type aliases, dynamic types comparison should not allow to distinguish aliases.

```
type a = A of int
type b = a list
let _ = Ty.eq (type b) (type a list) (* true *)
```

Abstracted type aliases have external identities

```
module M : sig
  type a
end = struct
  type a = int list
end
let _ = Ty.eq (type a) (type int list) (* false *)
```
Types in functor parameters could not be identified by an absolute path; by default, one may not build its dynamic type representation.

```ocaml
double X_0 = 2.0
```

Explicit annotation allows type in functor parameter to use the dynamic representation of the effective parameter:

```ocaml
double X_1 = 2.0
```

```ocaml
module F(X: sig type t end) = struct
  let t_repr = (type X.t) (* do not compile *)
end

module M1 = F(struct type t = int end) (* true *)
module M2 = F(struct type t = bool end) (* false *)
```
Types in unpacked first-order module could not be identified by an absolute path; by default, one may not build its dynamic type representation.

```ocaml
module type S = sig type t val x : t end
let f (module M : S) =
  print (type M.t) M.x (* Do not compile *)
```

Explicit annotation allows first-class module to pack the dynamic representation of type:

```ocaml
module type S' =
  sig type <transparent> t val x : t end
let f (module M : S') = print (type M.t) M.x
let _ = f (module struct type t = int end)
```
Conclusion

Work in progress...

Current prototype is available online:

http://gitorious.org/ocaml-ty/pages/Home