Runtime types in OCaml

Jacques Garrigue Grégoire Henry

1 Introduction

Following a proposition from Alain Frisch, we presented with Pierre Chambart at OUD 2012, a prototype extension of the OCaml compiler, (re)introducing runtime-types as first-class values in the language. Introspection of runtime-types in a type-safe manner was made possible by using a GADT describing the structure of the head type constructor. This was allowing one form of generic programming known as polytypic programming, a.k.a. functions defined by cases on the structure of their arguments types.

With this purely structural approach of runtime types, abstract types do not have a canonical representation. To allow the extension of a polytypic function with an *ad hoc* behaviour for a given abstract type, we relied on the manual creation and transmission of type witnesses. This approach has two main drawbacks:

- even with a mechanism for implicitly propagating type witnesses, manipulating abstract types involves a programming overhead
- as the representation of types is not unique, two parts of a program may end up defining different witnesses for the same type, hindering communication between them

We would like to present a new approach which makes nominal introspection of types more pervasive in the language, while keeping all the structural information required for polytypic programming. This relies on a canonical representation for all types, *i.e.* including abstract ones. We have a prototype implementation that works well for types in the *global context—i.e.* types in imported `.cmi` files. Sadly, there is no trivial notion of runtime-type equality when we want to compare, for instance:

- the internal concrete representation of a type and its external abstract representation; or,
- an abstract type in a functor parameter and its concrete instance in a functor application.

We would like to present to the community the different semantics for equality that we have identifed and the corresponding pratical consequences in terms of programming style, execution cost, and preservation of the abstraction. We are looking for feedback from potential users.

In the following, we try to illustrate the main difficulties by some examples.

2 A canonical representation

Consider the following interface, contained in `A.mli`.

```
module Id : sig
  type t
  val next : unit -> t
  val print : t -> unit
end

module type TREE = sig
  type t
  type id
  val leaf : id -> t
  val node : id -> t list -> t
  val print : t -> unit
end
module Tree = functor (X : module type of Id) ->
                      TREE with type id = X.t

module TreeId : TREE with type t = Tree(Id).t
                      and type id = Id.t
```

Without adding any extra information to `A` itself, we can build runtime representations for all types in this module. All we need is a global identifier `IdA`, corresponding to the compilation unit `A`. Then `A.Id.t` can be represented by the path `IdA.Id.t`. Type aliases have no proper identity, and are just represented by their contents, so that `A.TreeId.id` is again `IdA.Id.t`. We also respect the applicativity of functors, and represent `A.TreeId.t` by `IdA.Tree(IdA.Id).t`, so that applying `A.Tree` to `A.Id` in another compilation unit would still generate the same runtime representation.

While it incurs no cost, this representation is sufficient to handle all types exported by compilation units, preserving all the statically known type equalities, as long as we refer to them *from other compilation units.*

3 Abstraction

Consider the implementation file `A.ml`, where:

- the expression `(type t)` represents the runtime representation of a type `t`; it has the static type `t ty`;
- the `print` function in the `Tree` functor is implemented with a call to a generic printer of type: α `ty` $\rightarrow \alpha \rightarrow$ `unit`;
- the call to `Generic.register_printer` in the `Id`

module extends the generic printer with a printer for the abstract type `Id.t`.

```
module Id = struct
  type t = int
  let next =
    let cpt = ref 0 in
    fun () -> incr cpt; !cpt
  let print id = sprintf "id(%d)" id
  let () =
    Generic.register_printer (type t) print
end
module type TREE = ...
module Tree (X : module type of Id)
     : TREE with type id = X.t = struct
  type id = X.t
  type t =
    | Leaf of id
    | Node of id * t list
  let leaf id = Leaf id
  let node id l = Node (id, l)
  let print = Generic.print (type t)
end
module TreeId = Tree(Id)
```

Internal vs. external representations In this small example, we purposefully introduced two contradictory uses of the `(type t)` expression. For the first occurence, it should denote the abstract type `A.Id.t` and not the internal type `t` that is equal to `int`; otherwise, the custom printer would replace the default printer for all integers instead of being associated with the global abstract type `A.Id.t`. Meanwhile, for the second occurence of the expression `(type t)`, we want it to denote the concrete definition of `t`—which contains all the structural information about leafs and nodes—and not the global abstract one `A.Tree(X).t`; otherwise, the generic printer would not have enough information to print the value. In others words, when refering to types defined *in the current compilation unit*, one may have to choose between multiple possible representations of the same type.

Furthermore, while a runtime type-equality predicate could safely consider all these possible representations as equal when the comparison take place inside the compilation unit, this should probably not be the case when the comparison take place outside the compilation unit. Indeed, if an internal type representation implicity escapes its compilation unit, this would allow to dynamically break the abstraction.

Functor parameters For the `A.TreeId.print` function to have the expected behaviour, the type representation applied to the generic printer must contain all the required type information, including the concrete instance of the `X.t` type. Otherwise the generic printer would not be able to use the registered custom printer for `Id.t`. For that purpose, one possibilty is to *transparently* pass the runtime representation of its type parameter to a functor.

However, in order not to break the abstraction permitted by functor parameters, one may also prefer the runtime representation of `X.t` not to be related with its concrete instance. This means introducing a distinct anonymous representation of `X.t` for each functor application. Then, the expected behaviour for `A.TreeId.print` could be achieved with the following implementation:

```
module Tree (X : module type of Id)
     : TREE with type id = X.t = struct
  type id = X.t
  type t = ...
  (* ... *)
  let () =
    Generic.register_printer (type X.t) X.print
  let print = Generic.print (type t)
end
```

However, such anonymous representations break the canonicity property of runtime types in the global context.

Implicit aliases While in general we want to preserve type abstraction at runtime, there are some situations however where we may prefer not to. Typically, when the abstraction is not really used as a mechanism for preserving an invariant in a data structure but only as some sort of simplification mechanism. For instance, in the previous interface `a.mli`, the type equality `TreeId.t = Tree(Id).t` — which is implicit in the implementation file — is usually omitted for simplicity:

```
module TreeId : TREE with type id = Id.t
```

This introduces a new abstract type `A.TreeId.t`, that is different from the (already abstract) type `A.Tree(Id).t`. Then, if the `Tree` functor registers a custom printer for trees, it will be associated to the latter type only.

While a programmer can easily fix this situation by exporting the equation in the signature or explicitely registering the custom printer for `A.TreeId.t` after the functor application, this forces a more cumbersome programming style.

4 Conclusion

As we hope to have made clear in this proposal, handling the nominal aspect of runtime types in presence of abstraction is not an easy task. This is still work in progress, but we have already developed some solutions to the problems described. In the presentation we will explain the ideas underlying those solutions.