LexiFi Runtime Types

Patrik Keller[1] and Marc Lasson[2]

[1]University of Innsbruck
[2]LexiFi

28 August 2020

OCaml programmers make deliberate use of abstract data types for composing safe and reliable software systems. The OCaml compiler relies on the invariants imposed by the type system to produce efficient and compact runtime data representations. Being no longer relevant, the type information is discarded after compilation. The resulting performance is a key feature of the OCaml language.

However, the removal of type information during compilation also has downsides, e. g., it prohibits introspection. Having at hand the type of functions arguments at runtime, a programmer could implement generic serializers, parsers, database interfaces, and even GUIs.

In order to implement such generic functionality, OCaml programmers came up with a number of workarounds. Most notably, prepocessors can be used to derive non-generic functions for each relevant type. Preprocessing on the abstract syntax tree (AST), like it is possible with OCaml's PPX system, is very versatile. Probably any introspective or generic function could be replaced with this approach. However, writing PPX derivers is a tedious task, and, since AST rewriting precedes type inference, also prone to error.

Another approach is to use PPX *once* per type for deriving a runtime representation. The runtime type can then be used to write generic functions within usual OCaml programs. This approach is taken by, e. g., Irmin[1] and Janestreet[2].

LexiFi's approach is similar, but instead of relying on PPX, LexiFi maintains a compiler extension that derives runtime types using the OCaml typechecker. This allows us to synthesize the representation of inferred types without additional input from the programmer.

Recently, we implemented a PPX syntax extension that enables the use of LexiFi's runtime types on vanilla OCaml compilers. This will allow us to share LexiFi libraries that rely on runtime types with the OCaml community. As a first step, we provide the PPX and accompanying core libraries online[3].

LexiFi runtime types are composed of multiple components: An untyped representation (`type stype`) with a typed sibling (`type 'a ttype = stype`), a typed representation for safe introspection (`type 'a xtype`), a compiler extension (unreleased) that produces `'a ttype` and hands them to generic functions where

[1]`https://github.com/mirage/irmin/blob/master/README_PPX.md`
[2]`https://github.com/janestreet/ppx_typerep_conv`
[3]`https://github.com/LexiFi/lrt`

necessary, a PPX deriver that produces `'a ttype` on an unmodified compiler, and a (preliminary) unification mechanism that enables pattern matching on runtime types with holes.

After giving a brief overview on these components, our presentation will focus on the last aspect, namely pattern matching on runtime types. We will demonstrate why pattern matching on runtime types is useful, how it could be implemented, and what problems remain.

The following listing provides a small example for using our proof-of-concept implementation. We instantiate the pattern matcher in order to provide generic printing functionality. In the beginning, the matcher is empty. It does not understand any type and returns `"<opaque>"` independent of its input (Lines 14 and 15). After registering cases for the types `int` and `'a list`, the pretty printer can handle values of type `int` and `int list` (Lines 34 and 35).

```
1  let to_string_matcher = ref Matcher.empty
2
3  let to_string : type a. t : a ttype -> a -> string = fun ~t x ->
4    let open Matcher in
5    match apply !to_string_matcher with
6    | None -> "<opaque>"
7    | Some (M0 (module M : M0 with type matched = a)) -> M.return x
8    | Some (M1 (module M : M1 with type matched = a)) -> M.return x
9    | Some (M2 (module M : M2 with type matched = a)) -> M.return x
10
11 let p = print_endline
12
13 let () =
14   p (to_string ~t:[%t int] 42);              (* <opaque> *)
15   p (to_string ~t:[%t int list] [0; 1; 2]); (* <opaque> *)
16
17   (* register int to string conversion *)
18   to_string_matcher :=
19     Matcher.add ~t:[%t int] Int.to_string !to_string_matcher;
20
21   p (to_string ~t:[%t int] 42);              (* 42 *)
22   p (to_string ~t:[%t int list] [0; 1; 2]); (* <opaque> *)
23
24   (* register generic list to string conversion *)
25   to_string_matcher :=
26     Matcher.add1 (module struct
27       type 'a t = 'a list [@@deriving t]
28
29       let return t l =
30         List.map (to_string ~t) l
31         |> String.concat ", "
32       end) !to_string_matcher
33
34   p (to_string ~t:[%t int] 42);              (* 42 *)
35   p (to_string ~t:[%t int list] [0; 1; 2]); (* 0, 1, 2 *)
36   p (to_string ~t:[%t float list] [0.; 1.]) (* <opaque>, <opaque> *)
```

In the background, `Matcher.add` and its variants register each case in a discrimination tree index. The index enables efficient lookup of registered cases in `Matcher.apply` and does unification (e. g., filling the hole in the `'a list` converter with `int` and `string` as needed) on the fly.

We hope that our presentation encourages a fruitful discussion on runtime types and their effective use in OCaml.