Experiments in generic programming: runtime type representation and implicit values

Pierre Chambart & Grégoire Henry

June 7, 2012

Abstract

We present two ongoing related experiments in generic programming with OCaml. In the first experiment, inspired from the *implicit values* of Scala, the language is extended with primitives to enrich a locally scoped environment of implicit values—mapping types to values—and a construction for synthesising a value of a given type from the environment of implicit values. We have used this extension to implement light-weight, semi-implicit type classes and we are currently experimenting with different query mechanisms and other applications.

The second experiment is based on explicit *runtime-type representation*, described by a GADT, that can be analysed in a type-safe manner to implement polytypic functions, defined by cases on the structure of the types of their arguments.

Both experiments share a common set of non-intrusive extensions of the OCaml compiler; in particular, the modification in the type checker are simple, limited, and identical in both cases. This is because implicits can be implemented modularly as an additional phase in the compilation chain that comes right after type-checking.

1 Implicit value

The first experiment[1] is based on the notion of *implicit parameters* of a function, which can be automatically synthesised and inserted by the compiler from their types at their call-site.

The synthesis of an implicit parameter is based on its instantiated type and an environment of implicit values. For instance, the function `coerce`, defined below, takes an implicit parameter `f` of type `t1 -> t2`—introduced by the syntax `?#`—a regular parameter `x` and returns the application of `x` to `f`:

```
let coerce (type t1) (type t2) ?#(f : t1 -> t2) x = f x
```

The environment of implicit values may be enriched using a new syntactic construct `let implicit` p `=` e_1 `in` e_2 that behaves as `let` p `=` e_1 `in` e_2 but also

[1]Prototype: `https://gitorious.org/ocaml-ty/ocaml-implicit`

tells the compiler that (the value resulting from the evaluation of) e_1 may be used to build implicit values inside e_2. The implicit environment follows the usual scoping rules; we also use the toplevel form `let implicit` p `=` e_1 whose scope extends to the end of the current module. For instance, the following program will display "`3.3`".

```
let implicit _ = float_of_int
let implicit _ = string_of_float
let _ = print_string (coerce (1.3 +. coerce 2))
```

Parametrised implicit value The environment of implicit values may contain constant values, as in the example above, or functions constructing implicit values from implicit parameters. For instance, the following implicit function that builds a printer for lists is parametrised by a printer for its elements:

```
let rec implicit print_list :
  type a. ?#print_item:(a -> string) -> a list -> string =
  fun ?#print_item xs ->
    match xs with
    | [] -> "[]"
    | x :: xs -> print_item x ^ "::" ^ print_list xs
```

When this function is called implicitly to build a value of type `'a list -> string`, its implicit parameter is recursively synthesised. For instance, in the following program, the synthesised parameter for the function `coerce` is `print_list print_float` and the execution of the program displays "`1.::2.::[]`":

```
let _ = print_string (coerce [1.;2.])
```

Light-weight type classes Implicits can be used to encode type-class dictionaries represented as either records or first-order modules. However, contrary to Haskell, abstraction over dictionaries representing instance of type parameters must be explicitly introduced as extra arguments of functions. Still, the appropriate instances are automatically chosen at call-sites, which is the main purpose of type classes. Type-class inheritance may be encoded with parametrised implicit values.

Other usages of implicits We are currently experimenting with different strategies for querying the environment of implicit values, such as backtracking, using the instantiation ordering or a notion of cost of the synthesised values, *etc.*, and applications exploring the interest of the different strategies.

2 Runtime type representation

Implicit values allow some kind of generic programming based a on finite set of explicitly registered values. Hence, to benefit from this feature, a programmer must explicitly register type-class instances for each new datatype. Although we could automatically generate those instances from annotations on type definitions, we are also experimenting an alternative solution based on runtime type representation[2] as suggested by Alain Frish[3].

Runtime types allow the definition of generic functions defined by cases on the structure of types, much as implicit values. However, by contrast to implicit values, they can handle yet undefined types without further action by the programmer.

Head type constructors Runtime types are kept abstract for safety. To manipulate them, a library exports a GADT representing the head type constructor of a runtime type and provides functions to introspect or construct values of a given head type constructor. For instance, the following `print` function inspects the implicit argument `ty`—a runtime type representing the type `a`— for converting its second argument `v`—of type `a`—to a string.

```
let rec print (type a) ?#(ty : a ty) (v:a) =
  match Dyntypes.head ty with
  | Dyntypes.Int -> string_of_int v
  | Dyntypes.String -> v
  | ... (* Other base types *)
  | Dyntypes.Sum var ->
      let name, DynT (tup, args) = var.var_proj v in
      if List.length tup.tuple_field = 0 then name
      else name ^ "(" ^ print_fields tup.tuple_field args ^ ")"
  | ... (* Record, tuples, array. *)
and rec print_fields tys values =
  match tys, values with
  | [], [] -> ""
  | [ty], [v] -> print ~ty v
  | ty::tys, v::vs -> print ~ty v ^ "," print_fields tys vs
  | _, _ -> assert false
```

Revealing the representation of abstract types By default, our prototype is not able to synthesise runtime representation for abstract types. However, we provide a mechanism that allows the programmer of a library to define a custom runtime representation for an abstract type, and our synthesis mechanism is able to look up in the environment of implicit values for the custom representation of an abstract type.

[2]Prototype: `https://gitorious.org/ocaml-ty/ocaml-ty`
[3]Initial proposition: `http://www.lexifi.com/blog/runtime-types`

Currently, the only allowed representation for abstract types are opaque types or pairs of coercion functions towards a concrete type. For instance, the latter representation can be handled in the `print` function defined above by adding the following case:

```
    | Coerce (ty, concrete_of_abstract, abstract_of_concrete) ->
        print ~ty (concrete_of_abstract v)
```

The opaque representation can be used as key in an association table to define generic functions that can be later extended with specific behaviours for some opaque types.

Type variables By default, our prototype refuses to generate runtime type representations for non-closed types. However, one can still replace type variables with scoped abstract types and explicitly abstract their runtime type representations, as in the following examples:

```
  let print_list (type a) ?#(a_repr : a ty) (l : a list) =
    print l
```