Experiments in generic programming: runtime type representation and implicit values

Pierre Chambart[1], Grégoire Henry[2]

OUD 2012

[1]IRILL, CNRS, OCamlPro
[2]IRILL, PPS, Paris-Diderot

Every one complained about

```
let a = string_of_int 1
let b = string_of_float 3.14
```

rather than

```
let a = to_string 1
let b = to_string 3.14
```

Asking the compiler to code for us

Let's add a new type/value environment:

add_to_the_implicit_environment string_of_int

(find_me_a_value_of_type int $\rightarrow$ string)

Asking the compiler to code for us

Let's add a new type/value environment:

let implicit f = string_of_int

(**val of type** int → string)

replaced at compile time by f

First try

```
let implicit _ = string_of_int

let s = "the value is: " ^ ( (val of type _) 1 )
# val s : string = "the value is: 1"

let s = ( (val of type _) 1 )
# Error: Free variables in type (int → α)
```

Back to the example

We wanted to write

```
to_string 1
to_string 3.14
```

now we can do

```
let to_string f v : string = f a

to_string (val of type _) 1
to_string (val of type _) 3.14
```

Back to the example

```
let to_string ?#f a :string = f a
```

```
to_string 1
```

what the compiler see:

```
to_string ~f:(val of type _) 1
```

Simple but limited ?

```
let implicit string_of_list ?#string_of_elt l =
  "[ " ^
    String.concat "; " (List.map string_of_elt l)
  ^ " ]"

let s = to_string [1; 2]
# val s : string = "[ 1; 2 ]"
```

Simple but limited ?

looking for: (**val of type** int list $\rightarrow$ string)

In the environment:

string_of_int : int $\rightarrow$ string
string_of_list : ?#string_of_elt:(α $\rightarrow$ string) $\rightarrow$ α list $\rightarrow$ string

```
let s = to_string
  ~f:(string_of_list ~string_of_elt:string_of_int)
  [1; 2]
```

No black magic

There are no hidden type informations:

```
let stringify a = to_string a;;
# let stringify a = to_string a;;
# Error: Free variables in type (α → string)
```

No black magic

There are no hidden type informations:

```
let stringify a = to_string a;;
# let stringify a = to_string a;;
# Error: Free variables in type ('a -> string)
```

```
let stringify (f:'a -> string) (a:'a) =
  let implicit _ = f in
  to_string a
```

or

```
let stringify ?#(f:'a -> string) (a:'a) = to_string a
```

Not enough discipline

```
let implicit f i = string_of_int i
let implicit g i = String.make i ' '

(val of type int → string)
```

Not enough discipline: let restrict a bit

We only allow type constructors

```
type α stringable = {
  to_string : α → string;
  print : α → unit
}

let implicit _ =
  { to_string = int_of_string;
    print = print_int }

let to_string ?#f a = f.to_string a
```

For you haskellers

We are close to typeclasses:

```
class Eq a where
  equal :: a → a → Bool
class (Eq a) ⇒ Num a where
  plus :: a → a → a

type α eq = { equal : α → α → bool }
type α num = {
  num_eq : α eq;
  plus : α → α → α;
}
let implicit eq_of_num ?#num = num.num_eq
```

We are not restricted to records

```
class int_eq = object
  method eq (x:int) y = x = y
end

let eq ?#f (a:'a) (b:'a) : bool = f a b
let b = eq 1 2
# val b : bool = false
```

What we get

- Clean overloading on numbers, collections, etc ...
- What haskellers use typeclasses for

That's not all

Some heavy things stay heavy:

```
type big_record = { big record content }
to_string (v:big_record)
Error: Don't know how to build a value of
       type: big_record stringable
```

You still need to define a printer

Runtime type representations

A new special type: α ty

```
let int_representation = (val of type int ty)
# val int_representation : int ty = <abstr>
```

Runtime type representations

A new special type: α ty

```
let int_representation = (val of type int ty)
# val int_representation : int ty = <abstr>

head (val of type int ty)
# int head = Int
```

Runtime type representations

A new special type: α ty

```
let int_representation = (val of type int ty)
# val int_representation : int ty = <abstr>

head (val of type int ty)
# int head = Int

type _ head =
  | Int: int head
  | Float: float head
  | Array: α ty → α array head
  | ...
  | Abstract: ...

val head: α ty → α head
```

(Hooray for GADT)

Generic functions

```
val to_string_bis : α ty → α → string

let to_string_bis ty v = match head ty with
  | Int → string_of_int v
  | String → v
  | ...
```

- Difficult code: GADTs
- But you (someone else) only have to write it once

Usages

- Generic functions: printing, serialisation, ...
- Dynamic typing:

  type dyn = Dyn : α ty * α $\rightarrow$ dyn
- get rid of many camlp4 extensions

Way to upstream ?

- simple patch: mainly adds in separated parts
- safe: no modification/interraction with type checking
- implicits are globaly stable and complete
- the GADT for α head type is stable

Work in progress

- syntax is not fixed: it has changed, it *will* change
- using generic functions should be easy
  - usable by beginners: as easy as python ?
  - no type annotation ?
  - difficult to write generic functions is ok
- It is a generic framework to generate values from types: Other ideas ?
- Prolog like search mechanism: other interesting ?
- more testing
  - what will be future common paterns
  - what are meaningful restrictions
  - what are limiting restrictions
- https://gitorious.org/ocaml-ty/ocaml-ty
- https://gitorious.org/ocaml-ty/ocaml-implicit

Using dynamics

```
type dyn = Dyn: α ty * α → dyn

let cast_int dyn = match dyn with
  | Dyn (ty,v) →
    match head v with
      | Int → Some v
      | _ → None
```

Generic printer

```
let rec print (type a) ?#(ty : a ty) (v:a) =
  match head ty with
    | Int → string_of_int v
    | String → v
    | ... (* Other base types *)
    | 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
```

type representation and abstract types

```
val new_opaque: ?repr:α ty → unit → α ty

type t (* abstract *)
let implicit _ = (new_opaque ():t ty)
```