Type-safe type reflection for OCaml

Thierry Martinez

Cambium seminar, 9 november 2020
Initial motivation

Substituting the usage of `ppx_deriving` and `visitors` in `clangml`.

- `clangml` is an OCaml library of bindings for the Clang API.

- `clangml` exposes Clang’s AST for C/C++ as algebraic data types (≈400 types and ≈3,000 constructors).

- In `clangml`’s unit tests, values of these data types are compared (for equality, with `equal` plugin) and printed (with `show` plugin).

- In the MemCAD static analyser [Xavier Rival et al.] which uses the `clangml` library, values are compared (for sets and maps, with `compare` plugin) and visited and transformed (with various visitors [François Pottier]: `map`, `fold`, etc.).
Why type reflection?

There is a lot of functions that can be systematically derived from the structure of a given data type (for printing, comparing, visiting, serializing, etc.).

This set of derivable functions is

- open,
- depends on the usage of the data type,
- may add their own dependencies,
- and cannot be anticipated.

Reflecting types into values enables:

- type-safety: generic functions are type-checked instead of failing on random type declarations,
- modularity: the definition of generic functions and type declarations are orthogonal.
Related work

Thanks Nicolás Ojeda Bär for listing them by email one week before this talk!

- https://github.com/LexiFi/lrt
- https://github.com/janestreet/typerep
- https://github.com/mirage/repr

Portages of the “Scrap Your Boilerplate” Haskell’s framework:

- https://github.com/yallop/ocaml-syb
- https://github.com/pqwy/tpf
Basic idea: a GADT for discriminating types

Excerpt from lrt/lib/xtype.mli:

```ocaml
and 'a xtype = private
 | Unit : unit xtype
 | Bool : bool xtype
 | Int : int xtype
 [...]
 | List : 'b t -> 'b list xtype
 | Option : 'b t -> 'b option xtype
 | Array : 'b t -> 'b array xtype
 [...]
 | Record : 'a record -> 'a xtype
 | Sum : 'a sum -> 'a xtype
 | Function : ('b, 'c) arrow -> ('b -> 'c) xtype
 [...]
```
Roadmap: need to reflect the type *structure* in the GADT

1. 'a xtype reflects the *closed* type 'a, whereas most generic functions are expressed for type *constructors*. For a given type constructor 'a t, ppx_deriving can derive for instance:
   - show: ('a -> string) -> 'a t -> string
   - compare: ('a -> 'a -> int) -> 'a t -> 'a t -> int
   - map: ('a -> 'b) -> 'a t -> 'b t

2. some generic functions are not defined for some type constructions (for instance, compare may reject arrow types)

3. local variables can be introduced either universally (polymorphic fields) or existentially (GADTs). Moreover we need to keep track of the type equalities to satisfy (for GADTs and opaque types).
A data-structure to encode types

type (  
1. 'a: the type being reflected,  
2. 'structure: the structure/shape of the type, identical between instances of a given type constructor  
3. 'arity: a type-level list of the type constructor parameters:  
   'arity = ('a1 * (... ('an * unit))) when  
   'a = ('a1, ..., 'an) t  
4. 'rec_group: a type-level list of the reflection of the type constructors appearing in the same type declaration group,  
5. 'kinds: an open polymorphic variant of all the “features” used in the type declaration (for instance, compare can put an upperbound disallowing arrows)  
6. 'positive, 'negative, 'direct: type-level set of type parameters that occur positively, negatively, and “directly” (not under an arrow)  
7. 'gadt: a type-level list of equalities to maintain between type parameters (for GADTs and opaque types).  
) desc = [...]
}
An example of signature using (...) desc: pp/show

```ocaml
module Printer = struct
  type 'a t = Format.formatter -> 'a -> unit
end

module Printers = Vector (Printer)

let rec pp :
  type a structure arity rec_group
  positive negative direct gadt .
  (a, structure, arity, rec_group, 'kinds,
   positive, negative, direct, gadt) desc ->
  (arity, direct) Printers.t -> a Printer.t =
  fun desc printers fmt x -> [...]

let show desc printers x =
  Format.asprintf "%a" (pp desc printers) x;;
```
Examples of usage of show

show [%refl: (string * int) list] [] ["a", 1; "b", 2];;

definitions

show [%refl: _ list] [Some Format.pp_print_string]
 ["a"; "b"; "c"];;

type ('a, 'length) vector =
| [] : ('a, [~Zero]) vector
| (::) : 'a * ('a, 'length) vector ->
     ('a, [~Succ of 'length]) vector [@@deriving refl];;

show [%refl: (_, _) vector]
     [Some Format.pp_print_string; None] ["a"; "b"; "c"];;
Type-level data structures: lists

- Type-level lists:
  \[
  \texttt{[]} = \texttt{unit} \quad \text{and} \quad \texttt{[a :: b]} = \texttt{[a]*[b]}
  \]

```plaintext
module type UnaryTypeS = sig
  type 'a t
end

module Sequence (T : UnaryTypeS) = struct
  type _ t =
  | [] : unit t
  | (::) : 'head T.t * 'tail t -> ('head * 'tail) t
end

[(e1 : 'a1 T.t); ...; (en : 'an T.t)] :
  ('a1 * (... ('an * unit))) Sequence(T).t
```
Type-level data structures: sets

Used in 'positive, 'negative, 'direct to represent sets of type parameters that occur positively, negatively, and “directly” (not under an arrow).

- Type-level Booleans:
  \[ \text{true} = [\`\text{Present}] \text{ and } \text{false} = [\`\text{Absent}] \]

- Type-level sets: list of Booleans

```haskell
module Vector (T : UnaryTypeS) = struct
type ('a, 'occurrence) item =
  | None : (_, [\`Absent]) item
  | Some : 'a T.t -> ('a, _) item

type ('sequence, 'occurrences) t =
  | [] : (unit, unit) t
  | (::) : ('head, 'occurrence) item *
    ('tail, 'occurrences) t ->
    ('head * 'tail, 'occurrence * 'occurrences) t
end
```
Another example of signature using (...) desc: map

module Mapper = struct
  type ('a, 'b) t = 'a -> 'b
end

module Mappers = SignedVector (Mapper)

let rec map :
  type structure a_struct b_struct a_arity b_arity
  rec_group kinds positive negative direct gadt .
  (a_struct, structure, a_arity, rec_group, kinds, 
  positive, negative, direct, gadt) desc ->
  (b_struct, structure, b_arity, rec_group, kinds,  
  positive, negative, direct, gadt) desc ->
  (a_arity, b_arity, positive, negative) Mappers.t ->
  (a_struct, b_struct) Mapper.t =
  fun a_struct b_struct mapping x -> [...]
Signed vector

module type BinaryTypeS = sig
  type ('a, 'b) t
end

module SignedVector (T : BinaryTypeS) = struct
  type ('a, 'b, 'positive, 'negative) item =
    | None : ('a, 'b, ['Absent], ['Absent]) item
    | P : ('a, 'b) T.t -> ('a, 'b, _, ['Absent]) item
    | N : ('b, 'a) T.t-> ('a, 'b, ['Absent], _) item
    | PN : (('a, 'b) T.t * ('b, 'a) T.t) ->
          ('a, 'b, _, _) item
  type ('a, 'b, 'positive, 'negative) t =
    | [] : (unit, unit, unit, unit) t
    | (::): ('a, 'b, 'positive, 'negative) item *
          ('aa, 'bb, 'positive_tail, 'negative_tail) t
          -> ('a * 'aa, 'b * 'bb, 'positive * 'positive_tail,
               'negative * 'negative_tail) t
end
Examples of usage of map

type 'a binary_tree =
  | Leaf
  | Node of {
      left : 'a binary_tree; label : 'a;
      right : 'a binary_tree }
  @[deriving refl];;
map [%refl: _ binary_tree] [%refl: _ binary_tree]
  [P string_of_int]
  (Node { left = Leaf; label = 1; right = Leaf });;
let () = let refl = [%refl: 'a -> 'a] in
  let f = Refl.map refl refl
    [PN (string_of_int, int_of_string)] succ in
assert (f "3" = "4")
let () = let refl = [%refl: 'a -> 'b] in
  let f =
    Refl.map refl refl
    [N int_of_string; P string_of_int]
    (fun x -> succ x) in
assert (f "3" = "4")
Computing type-level sets of occurrences for a type declaration

\[
\text{type } (\text{'a}, \text{'b}, \text{'c}) \ t = (\text{'a} \to \text{'b}) \ast \text{'c}
\]
\[
P(t) = \{t_1, t_3\}, \ N(t) = \{t_2\}, \ D(t) = \{t_3\}
\]

(*t_i* is the *i*th parameter of *t*, a.k.a. 'a, 'b or 'c)

\[
\begin{align*}
\mathcal{P}(t) &= \text{'Present}*\text{'Absent}*(\text{'Present}*\text{unit}) \\
\mathcal{N}(t) &= \text{'Absent}*\text{'Present}*(\text{'Absent}*\text{unit}) \\
\mathcal{D}(t) &= \text{'Absent}*\text{'Absent}*(\text{'Present}*\text{unit})
\end{align*}
\]

How to compute type composition while preserving separate compilation?

\[
\text{type } \text{'a u =}
\]
\[
((\text{'a}, \text{'a} \to \text{unit}, \text{'a}) \ t, \ (\text{unit}, \text{unit} \to \text{'a}, \text{unit}) \ t, \\
\text{unit}) \ t
\]

Separate computation of set of occurrences

\[
\text{type 'a u = } \left( \left( \text{'a, 'a -> unit, 'a} \right) t, (\text{unit, unit -> 'a, unit}) t, \text{unit} \right) t
\]

Set of paths for occurrences of 'a in t:

\[\{ t_1 \cdot t_1 , t_1 \cdot t_2 \cdot -, t_1 \cdot t_3 , t_2 \cdot t_2 \cdot + \}\]

(t_i is the i-th parameter of t, and + marks the right-hand side of an arrow, and − marks the left-hand side of an arrow.)

Existence of a positive (resp. negative, direct) occurrence of 'a in u is equivalent to the existence of a positive (resp. negative, direct) path and is expressible as a Boolean expression: for instance, \( t_1 \cdot t_2 \cdot - \) is a positive path if and only if \( t_1 \in P(t) \land t_2 \in N(t) \lor t_1 \in N(t) \land t_2 \in P(t) \).
Coding path Boolean expressions in types

Predicates $t_i \in P(t)$ are coded as Church Boolean:

```plaintext
type ('if_true, 'if_false) t_1_positive = 'if_true
type ('if_true, 'if_false) t_1_negative = 'if_false
type ('if_true, 'if_false) t_2_positive = 'if_false
type ('if_true, 'if_false) t_2_negative = 'if_true
```

Path Boolean expressions can be coded as if-then-else expressions where conditions are on the predicates $t_i \in P(t)$, which can in turn be expressed as types:

```plaintext
[f[t_1 \in P(t) \land t_2 \in N(t) \lor t_1 \in N(t) \land t_2 \in P(t)] =.

((('if_true,
    ((((('if_true, 'if_false) t_2_positive,
        'if_false)
    t_1_negative) as 'if_else)
    t_2_negative,
    'if_else)
    t_1_positive
```
Restricting for some kinds of type construction

```ocaml
type ([...], 'kinds, [...]) desc =
  | Variable : [...], [> `Variable], [...]) desc [...]
  | [...]
  | Arrow : [...], ([...], 'kinds, [...]) desc [..., [> `Arrow] as 'kinds, [...]]) desc [...]

can be useful for forward-compatibility of generic functions too.
```
Record fields with universal quantifiers

▶ Two dual functions “construct” and “destruct”.

destruct :
  ('a, 'structure, 'arity, 'rec_group, 'kinds, 
   'subpositive, 'subnegative, 'subdirect, 'gadt, 'count, 
   forall_destruct); 

construct :
  ('a, 'structure, 'arity, 'rec_group, 'kinds, 
   'subpositive, 'subnegative, 'subdirect, 'gadt, 'count, 
   forall_construct -> 'a; 

▶ The introduced variables are \[\text{'Absent}\] at direct position: if they appear in direct positions, the type is not inhabited and the contradiction can be used.
GADTs with existential quantifiers

- Two dual functions “construct” and “destruct”.
- The presence of existential variables is marked in the `'kinds` argument.
Accumulating equalities for GADTs and opaque types

```plaintext
and ('eqs, 'structure_eqs, 'kinds, 'gadt) constructor_eqs =
  | ENil:
  (unit, unit, 'kinds, 'gadt) constructor_eqs
  | ECons: {
    head: ([`Succ of 'index], 'gadt, 'eq, _) selection
    tail:
    (eqs, 'structure_eqs, 'kinds, 'gadt) constructor_eqs
  } ->
  (eq * eqs, 'index * 'structure_eqs,
  [> `GADT] as 'kinds, 'gadt)
    constructor_eqs
```
Conclusion

- Generic functions can be defined type-safe: the type-checker proves that they handle all the types accepted in the signature. They may not terminate however.
- Handle polymorphic fields and GADTs.
- Work in progress, cleaning need.