# Purity inference

Jacques Garrigue

January 15, 2020

The value restriction, even in its relaxed form, makes us lose polymorphism even in cases where all functions are pure. This is especially galling for combinator libraries, where code is built through function applications.

Can we do better than that? The value restriction is only a weakened form of the original strong/weak type variable distinction present in Standard ML 90. It is a radical one: we only keep weak variables, deeming strong ones not modular enough.

Reintroducing strong variables would certainly be theoretically be fine, but there are practical reasons for their demotion. Here we will argue for a coarser grain approach: distinguish between pure and impure bindings in the environment. Our soundness criterion is as follows: a binding can be marked as pure if all its polymorphic type variables would be strong using the original SML 90 weak/strong discipline. This also shows the limitation of this approach: while SML 90 allowed mixing strong and weak type variables in the same polymorphic type, we allow only one category.

$$s ::= \mathsf{p} \mid \mathsf{i} \qquad \mathsf{p} \wedge s = s \qquad \mathsf{i} \wedge s = \mathsf{i}$$

$$\Sigma \vdash x : s \quad (x : s) \in \Sigma \qquad \frac{\Sigma, x : \mathsf{p} \vdash M : s}{\Sigma \vdash \mathsf{fun}\ x \to M : s} \qquad \frac{\Sigma \vdash M_1 : s_1 \quad \Sigma \vdash M_2 : s_2}{\Sigma \vdash M_1\, M_2 : s_1 \wedge s_2}$$

$$\frac{\Sigma \vdash M_1 : s_1 \quad \dots \quad \Sigma \vdash M_n : s_n}{\Sigma \vdash (M_1, \dots, M_n) : s_1 \wedge \dots \wedge s_n} \qquad \frac{\Sigma \vdash M_1 : s_1 \quad \Sigma \vdash M_2 : s_2}{\Sigma \vdash (M_1; M_2) : s_1 \wedge s_2}$$

$$\frac{\Sigma \vdash M_1 : s_1 \quad \Sigma, x : s_1 \vdash M_2 : s_2}{\Sigma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : s_1 \wedge s_2} \qquad \frac{\Sigma, x : \mathsf{p} \vdash M_1 : s_1 \quad \Sigma, x : s_1 \vdash M_2 : s_2}{\Sigma \vdash \mathsf{let\,rec}\ x = M_1\ \mathsf{in}\ M_2 : s_1 \wedge s_2}$$

$$\frac{\Sigma, x : \mathsf{i} \vdash M_1 : s_1 \quad \Sigma, x : s_1 \vdash M_2 : s_2}{\Sigma \vdash \mathsf{let\,rec}\ x : \forall \alpha.\sigma = M_1\ \mathsf{in}\ M_2 : s_1 \wedge s_2} \qquad \frac{\Sigma \vdash r : s_1 \quad \vdash_{\mathsf{lab}} l : s_2}{\Sigma \vdash r.l : s_1 \wedge s_2}$$

$$\vdash_{\mathsf{lab}} l : \begin{cases} \mathsf{i} & \text{when } l \text{ is a polymorphic field} \\ \mathsf{p} & \text{otherwise} \end{cases}$$

Polymorphic fields inside patterns also make the expression impure.

$$\frac{\Sigma \vdash M_i : s_i \quad \vdash_{\mathsf{mut}} l_i : s'_i}{\Sigma \vdash \{l_1 = M_1; \dots; l_n = M_n\} : \bigwedge_i s_i \wedge s'_i} \qquad \vdash_{\mathsf{mut}} l : \begin{cases} \mathsf{i} & \text{when } l \text{ is a mutable field} \\ \mathsf{p} & \text{otherwise} \end{cases}$$

$$\frac{\Gamma; \Sigma \vdash M_1 : \tau_1 \quad \Sigma \vdash M_1 : \mathsf{p} \quad \Gamma, x : \forall \vec{\alpha}.\tau_1; \Sigma, x : \mathsf{p} \vdash M_2 : \tau_2}{\Gamma; \Sigma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}\ \vec{\alpha} \cap \mathsf{fv}(\Gamma) = \emptyset$$

This solution is implemented by my `purity-attribute` branch: `https://github.com/garrigue/ocaml/tree/purity-attribute`, with a twist: the purity inference is not done on the raw syntax tree, but on a typing derivation. The rules $\Sigma \vdash x : s$ and $\vdash_{\mathsf{mut}} l : s$ are modified so that $s = \mathsf{p}$ if the type of that specific instance of $x$ or of (the argument of) $l$ contains no type variables (which matches the SML 90 criterion, since this means that this contains no weak type variables). This avoids to track purity on ground types, but unfortunately this is not principal.

Typed examples are at: `https://github.com/garrigue/ocaml/blob/purity-attribute/testsuite/tests/typing-poly/purity.ml`.

One limitation of the current implementation is that simultaneous let-definitions (using `and`) are typed as though they were tuples in the above rules.

**Examples**

```
(* Basic *)

let some x = Some x;;                          (* constructor functions can be pure *)
val some : 'a -> 'a option [@@pure]
let a = some (fun x -> x);;                  (* pure applications can be generalized *)
val a : ('a -> 'a) option [@@pure]

let r = ref;;                                          (* impure things are not pure! *)
val r : 'a -> 'a ref
let c = r (fun x -> x);;                   (* relaxed value restriction still applies *)
val c : ('_weak1 -> '_weak1) ref
let c' = {contents=fun x -> x};;
val c' : ('_weak2 -> '_weak2) ref

(* Modules *)

module type T = sig val some : 'a -> 'a option [@@pure] end;;

module M : T = struct let some x = Some x end;;
module M : T

module M' : T = struct let some x = let r = ref x in Some !r end;;
Error: Signature mismatch:
       Values do not match:
         val some : 'a -> 'a option
       is not included in
         val some : 'a -> 'a option [@@pure]

(* Beware of polymorphism *)
type mkref = {mkref: 'a. 'a -> 'a ref};;                           (* polymorphic field *)

let f x = let r = x.mkref [] in fun y -> r := [y]; List.hd !r;;
val f : mkref -> 'a -> 'a                                        (* impure field access *)

let f {mkref} = let r = mkref [] in fun y -> r := [y]; List.hd !r;;
val f : mkref -> 'a -> 'a                                             (* impure pattern *)

(* Beware of polymorphic recursion *)
let rec f x = let r = mkref x [] in fun y -> r := [y]; List.hd !r
and mkref : 'a. mkref -> 'a -> 'a ref = fun x -> x.mkref;;
val f : mkref -> 'a -> 'a                                    (* call to mkref is impure *)
val mkref : mkref -> 'a -> 'a ref

let rec id : 'a. 'a -> 'a = fun x -> id x;;
val id : 'a -> 'a                              (* impure due to polymorphic recursion *)

let rec id : 'a. 'a -> 'a = fun x -> x;;
val id : 'a -> 'a [@@pure]                             (* pure since no recursive call *)
```