Objective Caml: the language and its type system

Xavier Leroy
INRIA Rocquencourt, France
`http://caml.inria.fr/`

Outline

- **Part 1: the core language (and its type system)**
- Part 2: objects and classes (and their type system)
- Part 3: the module system (and its type system)

Thesis: strong typing with type inference gives coherence to an otherwise fairly diverse language (functional and imperative and object-oriented and ...)

The core Caml language

Caml programs can be written in files and compiled separately as usual.

For the examples, we use the interactive Caml system:

```
~$ ocaml
        Objective Caml version 2.00

# 1+2 ;;

     - : int = 3
```

The system computes the type and the value of each expression entered.

Caml as a desktop calculator

```
1 + 2 * 3;;
     - : int = 7

let pi = 4.0 *. atan 1.0;;
     val pi : float = 3.14159265359

let square x = x *. x;;
     val square : float -> float = <fun>

square(sin pi) +. square(cos pi);;
     - : float = 1
```

`let` gives a name to the result of a calculation.

With one or several parameters, `let` defines a function.

Strong static typing with type inference

The system checks each phrase for type-correctness.

Almost no type declarations need to be provided by the user: the system infers unspecified types from the context.

Ex: in **`let square x = x *. x`**, `x` must have type **`float`** because it is used as argument to the float multiplication `*.`

Integers and floats are distinct types, with distinct operators: `+` and `*` operate on integers; `+.` and `*.` on floats.

```
1.0 * 2;;
     This expression has type float but is here used with type int
```


Recursive functions

Recursive functions are defined with **`let rec`** instead of **`let`**:

```
let rec fib n =
  if n < 2 then 1 else fib(n-1) + fib(n-2);;
     val fib : int -> int = <fun>

fib 10;;
     - : int = 89
```

Due to strict static scoping, the system cannot guess whether its a **`let`** or a **`let rec`**

`let f x = x + 1`	`let f x = x + 1`
`let f y = f(y * 2)`	`let rec f y = f(y * 2)`

Basic data types

As usual: int, float, booleans, characters, character strings.

```
(1 < 2) = false;;
     - : bool = false

'a';;
     - : char = 'a'

"Hello world";;
     - : string = "Hello world"
```


Data structures, 1: lists

Lists can be given in extension `[ a; b; c ]` or built from the empty list `[]` (nil) by adding elements in front using `::` (cons).

```
let l = ["è"; "vero"; "è"; "ben"; "trovato"];;
     val l : string list = ["è"; "vero"; "è"; "ben"; "trovato"]

"Si" :: "non" :: l;;
     - : string list = ["Si"; "non"; "è"; "vero"; "è"; "ben"; "tro-
vato"]
```

The type of a list is τ `list` where τ is the type of its elements (lists must be homogeneous).

No explicit allocation or deallocation of "cons" cells (garbage collection). No explicit handling of pointers (done by the compiler).

Pattern-matching on lists

Pattern-matching is Caml's main mechanism for inspecting and destructuring data structures.

List patterns have the exact same shape as list expressions, with identifier representing unspecified parts of the list:

- `[]` an empty list
- `a :: b` a non-empty list, with `a` bound to its first element and `b` its tail
- `[a; b; c]` a 3-element list, with `a`, `b`, `c` bound to those elements.

Insertion sort on lists

```
let rec sort lst =
  match lst with
    [] -> []
  | head :: tail -> insert head (sort tail)

and insert elt lst =
  match lst with
    [] -> [elt]
  | head :: tail ->
      if elt <= head then elt :: lst else head :: insert elt tail
;;
     val sort : 'a list -> 'a list = <fun>
     val insert : 'a -> 'a list -> 'a list = <fun>

sort l;;
     - : string list = ["ben"; "trovato"; "vero"; "è"; "è"]
```

Polymorphism

Notice the type inferred for `sort`:

```
val sort : 'a list -> 'a list
```

It means: "`sort` can be applied to lists of any type, and returns a list of the same type". The type `'a` is a type variable, and can be replaced by any given type.

```
sort [6; 2; 5; 3];;
    - : int list = [2; 3; 5; 6]

sort [3.14; 2.718];;
    - : float list = [2.718; 3.14]
```

We say that `sort` is *polymorphic*, and the reason it is polymorphic is because the comparison operators `=`, `<=`, etc. are themselves polymorphic: they operate between any two values of the same type.

Functions as values

Caml is a functional language: functions in the full mathematical sense are supported and can be passed around freely just as any other piece of data.

Example: approximating the derivative of a function.

```
let deriv f dx = function x -> (f(x +. dx) -. f(x)) /. dx;;
     val deriv : (float -> float) -> float -> float -> float = <fun>

let cos' = deriv sin 1e-6;;
     val cos' : float -> float = <fun>

cos' pi;;
     - : float = -1.00000000014
```

Functions taking or returning other functions are called functionals, or higher-order functions.

More functionals

Function composition:

```
let compose f g = function x -> f(g(x));;
     val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

let cos2 = compose square cos;;
     val cos2 : float -> float = <fun>
```

Generic operations over data structures (iterators):

```
let rec map f l =
  match l with
     [] -> []
   | hd :: tl -> f hd :: map f tl;;
     val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

map (function n -> n * 2 + 1) [0; 1; 2; 3; 4];;
     - : int list = [1; 3; 5; 7; 9]
```


User-defined data structures, 1: records

The `type` declaration lets the user define record types and variant (disjoint sum) types.

```
type ratio = { num: int; denum: int };;
     type ratio = { num: int; denum: int }

let add_ratio r1 r2 =
  { num = r1.num * r2.denum + r2.num * r1.denum;
    denum = r1.denum * r2.denum }
;;
     val add_ratio : ratio -> ratio -> ratio = <fun>
```

User-defined data structures, 2: variants

The declaration of a variant type lists all possible shapes for values of that type. Each case is identified by a constructor (capitalized name). The constructor serves both for constructing values of the variant type and inspecting them by pattern-matching.

Example: symbolic computation on arithmetic expressions.

```
type expression =
      Const of float
    | Var of string
    | Sum of expression * expression    (* e1 + e2 *)
    | Diff of expression * expression   (* e1 - e2 *)
    | Prod of expression * expression   (* e1 * e2 *)
    | Quot of expression * expression   (* e1 / e2 *)
```


Evaluation of an expression

```
let rec eval env exp =
    match exp with
      Const c -> c
    | Var v -> List.assoc v env
    | Sum(f, g) -> eval env f +. eval env g
    | Diff(f, g) -> eval env f -. eval env g
    | Prod(f, g) -> eval env f *. eval env g
    | Quot(f, g) -> eval env f /. eval env g;;

     val eval : (string * float) list -> expression -> float = <fun>
```

env is an association list (list of pairs) mapping variable names to their values.

```
eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
- : float = 9.42
```

Symbolic derivation of an expression

```
let rec deriv exp dv =
  match exp with
    Const c -> Const 0.0
  | Var v -> if v = dv then Const 1.0 else Const 0.0
  | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
  | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
  | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
  | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
                       Prod(g, g))
;;
     val deriv : expression -> string -> expression = <fun>

deriv (Quot(Const 1.0, Var "x")) "x";;
     - : expression =
        Quot
         (Diff (Prod (Const 0, Var "x"), Prod (Const 1, Const 1)),
          Prod (Var "x", Var "x"))
```


Imperative features

All examples so far are purely applicative (no in-place modification on data structures and no assignment to identifiers).

Caml is also equipped with full imperative power:

- Mutable data structures (modifiable in place by assignment).
- Sequence $a; b$
- Loops (`for`, `while`).
- Exceptions.
- Input/output by side effects.

Mutable data structures

Most Caml data structures are immutable (lists, sum types), but some are mutable (modifiable in place by assignment):

- Arrays τ **array**
- References (indirection cells; one-element arrays) τ **ref**
- Records declared with mutable fields

```
type point = { mutable x: float; mutable y: float };;

let translate p dx dy =
    p.x <- p.x +. dx; p.y <- p.y +. dy;;
```


Imperative programming: insertion sort on arrays

```
let insertion_sort a =
  for i = 1 to Array.length a - 1 do
    let val_i = a.(i) in
    let j = ref i in
    while !j > 0 && val_i < a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      j := !j - 1
    done;
    a.(!j) <- val_i
  done;;
     val insertion_sort : 'a array -> unit = <fun>

let a = [| 3.14; 1.2; 2.718 |];;
     val a : float array = [|3.14; 1.2; 2.718|]
insertion_sort a;;
     - : unit = ()
a;;
     - : float array = [|1.2; 2.718; 3.14|]
```

A simplified language

Expressions:

$$
\begin{array}{rll}
a ::= & c & \text{constants} \\
\mid & x & \text{identifiers} \\
\mid & \mathtt{fun}\ x \to a & \text{function abstraction} \\
\mid & a_1\ a_2 & \text{function application} \\
\mid & \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 & \text{the } \mathtt{let} \text{ binding}
\end{array}
$$

Constants c include base constants (0, 1, **`false`**, **`true`**, **`"hello"`**) and predefined operations (**`+`**, . . .).

Many derived forms, e.g. **`let f x y = a`** is
`let f = fun x` $\to$ **`fun y`** $\to$ **`a in`** *rest of session*

The type algebra

Type expressions:

$$
\begin{array}{rll}
\tau ::= & \mathtt{int} & \text{integers} \\
\mid & \mathtt{bool} & \text{booleans} \\
\mid & \ldots & \text{other base types} \\
\mid & \alpha & \text{type variables} \\
\mid & \tau_1 \to \tau_2 & \text{function types}
\end{array}
$$

The typing rules

Define the typing predicate $E \vdash a : \tau$
(read: under assumptions E, expression a has type τ)

E maps identifiers to their types.

The predicate is defined by a set of inference rules of the form

$$\frac{P_1 \qquad \ldots \qquad P_n}{P}$$

(read: if the assumptions $P_1, \ldots, P_n$ hold, then P holds)

Typing rules for the simply-typed λ-calculus

$$E \vdash x : E(x) \qquad\qquad E \vdash c : TC(c)$$

$$\frac{E \oplus \{x : \tau_1\} \vdash a : \tau_2}{E \vdash \texttt{fun}\ x \rightarrow a : \tau_1 \rightarrow \tau_2}$$

$$\frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau}$$

$$\frac{E \vdash a_1 : \tau' \qquad E \oplus \{x : \tau'\} \vdash a_2 : \tau}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

TC assigns types to constants, e.g. $TC(0) = \texttt{int}$,
$TC(+) = \texttt{int} \rightarrow (\texttt{int} \rightarrow \texttt{int})$.

Typing derivations

A typing derivation is a justification (a proof) of well-typing.

$$\dfrac{\dfrac{\dfrac{\{x : \texttt{int}\} \vdash + : TC(+) \qquad \{x : \texttt{int}\} \vdash 1 : \texttt{int}}{\{x : \texttt{int}\} \vdash +\ x : \texttt{int} \rightarrow \texttt{int}} \qquad \{x : \texttt{int}\} \vdash 1 : \texttt{int}}{\{x : \texttt{int}\} \vdash +\ x\ 1 : \texttt{int}}}{\emptyset \vdash (\texttt{fun}\ x \rightarrow +\ x\ 1) : \texttt{int} \rightarrow \texttt{int}}$$

Does this type system make sense?

A type system is *sound* if (compile-time) typings are consistent with the (run-time) evaluation of the program:

- A well-typed program must never go wrong at run-time by evaluating a nonsensical expression such as 1 2 or $+\ (\texttt{fun}\ x \rightarrow a)$.
- The final value of an expression must belong to the type assigned to the expression. E.g. if $E \vdash a : \texttt{int}$, then a must evaluate to an integer value.

We need to specify the evaluation of an expression.

Reduction semantics

$$a \xrightarrow{*} a_1 \xrightarrow{*} a_2 \xrightarrow{*} \ldots \xrightarrow{*} v$$

We keep rewriting the expression itself according to some evaluation rules until one of the following happens:

- We reach a normal form (an expression that cannot be reduced any further) and that normal form is a *value* v: the expression successfully evaluated to v (OK)
- We have an infinite reduction sequence: the computation does not terminate (OK)
- We reach a stuck expression (a normal form that is not a value): the evaluation went wrong (bad)

The reduction rules

Values: $v ::= c \mid \texttt{fun}\ x \to a$

Reduction rules:

$$\begin{array}{rcl} (\texttt{fun}\ x \to a)\ v & \xrightarrow{*} & a\{x \leftarrow v\} \\ \texttt{let}\ x = v\ \texttt{in}\ a & \xrightarrow{*} & b\{x \leftarrow v\} \\ +\ n_1\ n_2 & \xrightarrow{*} & n \text{ where } n = n_1 + n_2 \\ E[a] & \xrightarrow{*} & E[b] \text{ if } a \xrightarrow{*} b \end{array}$$

Reduction contexts E determine the order of evaluations (here, left-to-right call-by-value):

$E ::= [\,]$	reduction at head
$\mid E\ a$	in function part of an application
$\mid v\ E$	in argument part of a value application
$\mid \texttt{let}\ x = E\ \texttt{in}\ a$	in left part of a `let`

Soundness of typing w.r.t. evaluation

Reduction preserves typing: (*subject reduction*)
if $\emptyset \vdash a : \tau$ and $a \xrightarrow{*} b$, then $\emptyset \vdash b : \tau$.

Well-typed expressions are never stuck:
if $\emptyset \vdash a : \tau$ and $a \overset{*}{\not\to}$, then a is a value.

Corollary: **soundness of typing**
if $\emptyset \vdash a : \tau$, then a either reduces to a value v or loops, but never gets stuck.

Type inference

The typing rules allow to check the validity of a given type derivation.

What we need now is an algorithm that takes a program as input and checks that it is well-typed and infers the types of its expressions.

We view this as a constraint satisfaction problem:

- Associate a type unknown T_a to each subexpression a of the program and a type unknown T_x to each identifier x.
- Based on the typing rules, derive equality constraints between the T_a and T_x.
- Find a principal solution to the constraints.

Setting up the constraints

- If $a = x$: add $T_a = T_x$
- If $a = c$: add $T_a = TC(c)$
- If $a = \texttt{fun } x \to b$: add $T_a = T_x \to T_b$
- If $a = b(c)$: add $T_b = T_c \to T_a$
- If $a = \texttt{let } x = b \texttt{ in } c$: add $T_x = T_b$ and $T_a = T_c$

The constraints $\mathcal{C}$ capture all possible typings of the program in the following sense: $\{x : T_x\} \vdash a : T_a$ is derivable iff the T_x, T_a are solutions of $\mathcal{C}$.

Solving the constraints

The constraints are sets of equations between first-order terms (where the T_x and T_a are viewed as variables).

These constraints are solved by the standard unification algorithm between first-order terms.

Given a set $\mathcal{C}$ of constraints, the algorithm either

- fails, meaning that $\mathcal{C}$ has no solutions;
- or returns a subtitution φ assigning types to the T_x and T_a that is a principal solution of $\mathcal{C}$:
  - (correctness) φ is a solution of $\mathcal{C}$, i.e. for all equations $\tau_1 = \tau_2$ in $\mathcal{C}$, we have $\varphi(\tau_1) = \varphi(\tau_2)$
  - (principality) if ψ is a solution if $\mathcal{C}$, there exists θ such that $\psi = \theta \circ \varphi$.

An example of type inference

Consider $a = \texttt{fun}\ x \to \texttt{fun}\ y \to 1$ and $b = a\ \texttt{true}$.

We obtain the following equations:

$$T_a = T_x \to T_y \to T_1 \qquad T_1 = \texttt{int} \qquad T_a = T_{true} \to T_b \qquad T_{true} = \texttt{bool}$$

A principal solution is:

$$T_x = \texttt{bool} \qquad T_y = \alpha \qquad T_a = \texttt{bool} \to \alpha \to \texttt{int} \qquad T_b = \alpha \to \texttt{int}$$

All other solutions are obtained by replacing α by any type expression τ.

In summary

Combining constraint building and constraint solving, we obtain an algorithm $W(a)$ which either

- fails, meaning that a is ill-typed
  (there does not exist τ such that $\emptyset \vdash a : \tau$)
- or returns a type τ which is the principal type of a:
  - $\emptyset \vdash a : \tau$
  - if $\emptyset \vdash a : \tau'$, then $\tau' = \theta(\tau)$ for some substitution θ.

We say that the type system admits principal types.

Adding polymorphism

The current type system admits undetermined types (α) but not polymorphism: there is no way to use a function with several different types.

```
let f = fun x → x in
```

at this point, $\texttt{f} : \alpha \to \alpha$

```
let y = f 1 in
```

here we must take $\alpha = \texttt{int}$

```
f true
```

we get at type error as `int` and `bool` are incompatible

Type schemes

Type schemes: $\sigma ::= \forall \alpha_1, \ldots, \alpha_n.\ \tau$

Type schemes are a compact representation of all possible types of a polymorphic expression.

Example: $\forall \alpha.\ \alpha \to \alpha$ stands for the set of types $\{\tau \to \tau \mid \tau \text{ type}\}$, which are all acceptable types for $\mathbf{fun}\ x \to x$.

Instanciation relation: $\tau \leq \sigma$ (read τ is an instance of σ) if $\sigma = \forall \alpha_1, \ldots, \alpha_n.\ \tau'$ and $\tau = \tau'\{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}$ for suitable types $\tau_1 \ldots \tau_n$.

The polymorphic `let` rule

We allow `let`-bound identifiers to be assigned type schemes in the typing environment.

$$\frac{\tau \le E(x)}{E \vdash x : \tau}$$

$$\frac{E \vdash a : \tau_1 \qquad E \oplus \{x : Gen(\tau_1, E)\} \vdash b : \tau}{E \vdash \texttt{let}\ x = a\ \texttt{in}\ b : \tau}$$

$Gen(\tau, E) = \forall \alpha_1, \ldots, \alpha_n.\ \tau$ where $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau) \setminus FV(E)$.

(The α_i are the variables that occur in τ and that are generalizable at this point because they are completely unconstrained, i.e. they do not occur elsewhere in the typing environment E.)

Example of polymorphic typing

$$\frac{\dfrac{\{x : \alpha\} \vdash x : \alpha}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\texttt{int} \to \texttt{int} \le \forall \alpha.\alpha \to \alpha}{\{f : \forall \alpha.\alpha \to \alpha\} \vdash f : \texttt{int} \to \texttt{int}} \quad \{f : \forall \alpha.\alpha \to \alpha\} \vdash 1 : \texttt{int}}{\{f : \forall \alpha.\alpha \to \alpha\} \vdash f\ 1 : \texttt{int}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f\ 1 : \texttt{int}}$$

Results

The previous results extend to the system with `let`-polymorphism:

- Type soundness (subject reduction, etc)
- Existence of principal typings; the type inference algorithm.

  (Although now we need to interleave constraint building and constraint solving, so that `Gen` at `let` points is handled correctly.)

(See course handout for detailed proofs and algorithms.)

Extension 1: recursion

$$a ::= \ldots \mid \texttt{let rec } x = a \texttt{ in } b$$

Typing rule:

$$\frac{E \oplus \{x : \tau_1\} \vdash a : \tau_1 \qquad E \oplus \{x : Gen(\tau_1, E)\} \vdash b : \tau}{E \vdash \texttt{let rec } x = a \texttt{ in } b : \tau}$$

Notice that x must be used monomorphically in a; it is polymophic only in b.

```
let rec f x = f 1; x in f true;;
     This expression has type bool but is here used with type int
```

`f` has type `int` $\rightarrow$ `int` instead of $\forall\alpha.\ \alpha \rightarrow \alpha$ (which would be semantically correct) because `f` was used with type `int` $\rightarrow$ `int` in the recursion.

Polymorphic recursion

We can allow polymorphic uses of a recursively-bound identifier by "guessing" a type scheme for it rather than a simple type:

$$\frac{E \oplus \{x : \sigma\} \vdash a : \tau_1 \qquad \sigma = Gen(\tau_1, E) \qquad E \oplus \{x : \sigma\} \vdash b : \tau}{E \vdash \texttt{let rec } x = a \texttt{ in } b : \tau}$$

However, this renders type inference undecidable
(type inference then reduces to a semi-unification problem instead of a unification problem, and semi-unification is undecidable).

Extension 2: data types and pattern-matching

We treat constructors as new constants with associated type schemes. For instance:

```
type 'a list = Nil | Cons of 'a * 'a list
```

$$TC(\texttt{Nil}) = \forall\alpha.\ \alpha\ \texttt{list} \qquad TC(\texttt{Cons}) = \forall\alpha.\ \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}$$

We add pattern-matching:

Expressions: $a ::= \ldots \mid \texttt{match } a \texttt{ with } p_1 \to a_1 \mid p_n \to a_n$

Patterns: $p ::= \_ \mid x \mid c \mid c(p_1, \ldots, p_n)$

Typing `match` and patterns

$$\frac{E \vdash a : \tau' \qquad \vdash p_i : \tau' \Rightarrow E_i \qquad E \oplus E_i \vdash a_i : \tau \text{ for all } i = 1 \ldots n}{E \vdash \texttt{match } a \texttt{ with } p_1 \rightarrow a_1 \mid p_n \rightarrow a_n : \tau}$$

$\vdash p : \tau \Rightarrow E$ means "pattern p matches values of type τ and binds identifiers as described in E".

$$\vdash \_ : \tau \Rightarrow \emptyset \qquad\qquad \vdash x : \tau \Rightarrow \{x : \tau\} \qquad\qquad \frac{\tau \leq TC(c)}{\vdash c : \tau \Rightarrow \emptyset}$$

$$\frac{\tau_1 \times \cdots \times \tau_n \rightarrow \tau \leq TC(c) \qquad \text{for all } i = 1 \ldots n,\ \vdash p_i : \tau_i \Rightarrow E_i}{\vdash c(p_1, \ldots, p_n) : \tau \Rightarrow E_1 \oplus \cdots \oplus E_n}$$

Extension 3: references and assignment

We model mutable data structures with references (indirection cells).

τ **ref** is the type of references containing a value of type τ.

Operations:

- Create a new reference

$$\texttt{ref} : \forall \alpha.\ \alpha \rightarrow \alpha\ \texttt{ref}$$

- Dereferencing (access the contents of a reference)

$$\texttt{!} : \forall \alpha.\ \alpha\ \texttt{ref} \rightarrow \alpha$$

- Assignment (change the contents of a reference)

$$\texttt{:=} : \forall \alpha.\ \alpha\ \texttt{ref} \rightarrow \alpha \rightarrow \texttt{unit}$$

The problem with polymorphic references

When references are added, the monomorphic type system remains sound, but the polymorphic type system becomes unsound:

`let r = ref(fun x → x) in`	here, $\texttt{r} : \forall\alpha.\ (\alpha \to \alpha)\ \texttt{ref}$
`  r := (fun x → x + 1);`	we use `r` with type `(int→int) ref`; fine
`  (!r)(true)`	we use `r` with type `(bool→bool) ref`; fine

The program gets stuck at run-time trying to evaluate `true + 1`.

The value restriction

The simplest and most practical solution to the problem with polymorphic references is to restrict `let`-generalization to expressions that are syntactically values (i.e. constants and function abstractions):

$$\frac{E \vdash a : \tau_1 \qquad \sigma = \begin{cases} Gen(\tau_1, E) & \text{if } a \text{ is a value;} \\ \tau_1 & \text{otherwise} \end{cases} \qquad E \oplus \{x : \sigma\} \vdash b : \tau}{E \vdash \texttt{let}\ x = a\ \texttt{in}\ b : \tau}$$

This rules out the problem, since `ref` a is not a value.

The value restriction in practice

```
let f = fun x → y → y;;
```
$f : \forall\alpha,\beta.\ \alpha \to \beta \to \beta$

```
let g = f 1;;
```
$g : \gamma \to \gamma$

```
g 1;;
```
- : int

```
g;;
```
- : int → int

```
g true;;
```
Type error

```
let g = fun y → f 1 y;;
```
$g : \forall\gamma.\ \gamma \to \gamma$

```
g 1;;
```
- : int

```
g true;;
```
- : bool

Works well in practice because 99% of all useful polymorphic values are functions **fun** $x \to a$.