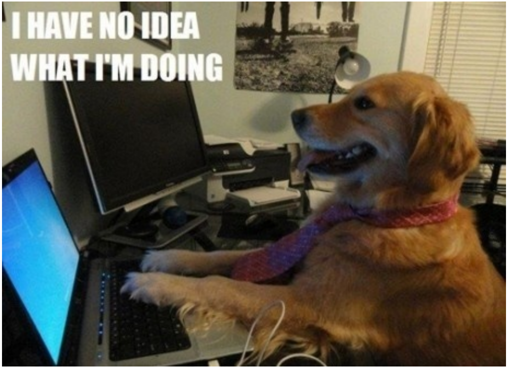
ArtiCheck

Thomas Braibant
Jonathan Protzenko
Gabriel Scherer

What is testing?



Let's make it a little bit smarter than that.

The basics of the library

Running example

```
(* tree.mli *)
type t
val empty: t
val add: t -> int -> t
val remove: t -> int -> t

val check: t -> bool
```

Running example

```ocaml
(* tree.mli *)
type t
val empty: t
val add: t -> int -> t
val remove: t -> int -> t

val check: t -> bool
```

We want to act *as a fake user* of the library.

Running example

```
(* tree.mli *)
type t
val empty: t
val add: t -> int -> t
val remove: t -> int -> t

val check: t -> bool
```

We want to act *as a fake user* of the library.

External testing *vs. internal* testing

Good call *vs.* bad call

Only *certain* calls are *well-typed*.

- **`add empty 1`** = GOOD
- **`add add add`** = BAD

Getting type-theoretic (1)

GADTs! Describing well-typed calls.

```
type (_, _) fn =
| Ret: 'a ty -> ('a, 'a) fn
| Fun: 'a ty * ('b , 'c) fn -> ('a -> 'b, 'c) fn
```

The type `('a, 'b) fn` describes a function with arrow type `'a`, whose return type is `'b`.

Getting type-theoretic (2)

Type descriptors.

```
type 'a ty = {
  mutable enum: 'a list;
  fresh: ('a list -> 'a) option;
}
```

The type **'a ty** describes a *collection of instances* for type **'a**.

- For **int**: **fresh** generates a fresh integer each time.
- For **t**: no **fresh** function.

Evaluating!

```
let rec eval : type a b. (a,b) fn -> a -> b list =
  fun fd f ->
    match fd with
    | Ret _ -> [f]
    | Fun (ty,fd) -> List.flatten (
        List.map (fun e -> eval fd (f e)) ty.enum)
let rec codom : type a b. (a,b) fn -> b ty =
  function
    | Ret ty -> ty
    | Fun (_,fd) -> codom fd
```

Registering new instances

```
let use (fd: ('a, 'b) fn) (f: 'a): unit =
  let prod, ty = eval fd f, codom fd in
  List.iter (fun x ->
    if not (List.mem x ty.enum)
    then ty.enum <- x::ty.enum
  ) prod
```

Declaring an interface

```
type sig_elem = Elem : ('a,'b) fn * 'a -> sig_elem
type sig_descr = (string * sig_elem) list

let tree_t : Tree.t ty  = ...
let int_t = ... (* integers use a [fresh] function*)

let sig_of_tree = [
  ("empty", Elem (returning tree_t, Tree.empty));
  ("add", Elem (tree_t @-> int_t @-> returning tree_t, Tree.add)); ]
let _ =
  Arti.generate sig_of_tree;
  assert (Arti.counter_example tree_t Tree.check = None)
```

Where the trouble begins