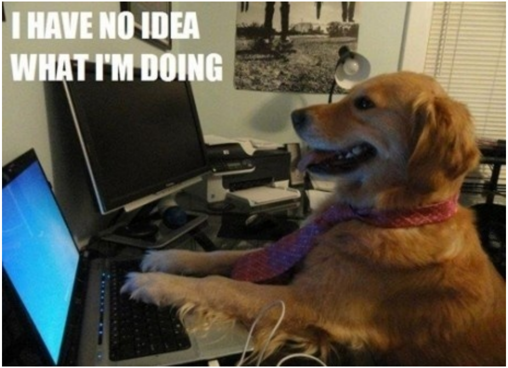
ArtiCheck

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

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

val check: t -> bool
```

Running example

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

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 option

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

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

Getting type-theoretic (1)

GADTs! Describing well-typed calls.

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

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

```
let add_fn: (Tree.t -> int -> Tree.t) fn =
  Fun (tree_ty, Fun (int_ty, Ret tree_ty))
```

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)
```

Registering new instances

```
let rec codom : type a b. (a,b) fn -> b ty =
  function
    | Ret ty -> ty
    | Fun (_,fd) -> codom fd

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)
```

More on the actual implementation (1)

- Two GADTs, **neg** for computation and **pos** for values.
- Destructuring: sums and products are composed and decomposed.

```
type (_, _) neg =
| Fun : 'a pos * ('b, 'c) neg -> ('a -> 'b, 'c) neg
| Ret : 'a pos -> ('a, 'a) neg
and _ pos =
| Ty : 'a ty -> 'a pos
| Sum : 'a pos * 'b pos -> ('a, 'b) sum pos
| Prod : 'a pos  * 'b pos -> ('a * 'b) pos
| Bij : 'a pos * ('a, 'b) bijection -> 'b pos
and ('a, 'b) sum = L of 'a | R of 'b
and ('a, 'b) bijection = ('a -> 'b) * ('b -> 'a)
```

More on the actual implementation (2)

- Symbolic representation of *sets* of values (e.g. **`Union`**, **`Product`**, etc.) to tame combinatorial explosion (don't build all possible products!).
- A fixpoint computation to implement the equations between types.

Where the trouble begins

Recap

- We describe a module interface.
- We act as a user of the interface, and construct instances of a given type.
- We do all of that in a well-typed manner.
- The library exports a **check** function: we do not access its internals.

Issue #1: exploration

The exploration functions describe:

- how to build new instances
- what to do with old instances

Example: breadth-first search.

Pseudo-code:

```
let bfs =
  List.map (fun f -> [ f x <- ∀ x ∈ domain(f) ]) all_functions
    |> sort_of_flatten
    |> register
```

Issue #1: exploration

- Discarding strategy: test predicate on the fly and remove "old" instances from round $n-1$. Is that right? (They may be useful later?) Too slow?
- Class of functions: what are other useful functions? Should the client be able to parameterize the search?
- Is there any way to skew the exploration towards more meaningful functions?

Issue #2: predicates

We offer a simple language of predicates.

$$\forall x_1 \ldots x_n, P(x_1, \ldots, x_n) \Rightarrow T(x_1, \ldots, x_n)$$

where $P(x_1, \ldots, x_n)$ (the *precondition*) and $T(x_1, \ldots, x_n)$ (the *test*) are both quantifier-free formulas.

Makes a difference between a meaningful test (precondition was met, predicate is false) and a meaningless test (precondition was not met).

Good? Overkill?

Issue #3: testcase reduction

We keep call trees that led to a failure, i.e. we keep witnesses of type 'a wit.

What is a good strategy for reducing the size of a witness? Should the user provide a function `'a wit -> 'a wit list` that breaks up a witness into smaller candidates? (e.g. exposes the sub-trees?)

We lose the connection to the original call tree. May not be dramatic for some use-cases (HSMs).

Issue #4: higher-order

Polymorphic (parameterized) types: monomorphize.

Higher-order functions: we do not synthesize, and ask the user to provide a few representative candidates. Is it foolish?

Issue #5: good examples

Data structures are cool. Thomas' work on HSMs is cool too. Anything else worth checking?

Issue #6: proper library

Auto-generate interface descriptions. Working on it.

Any other suggestions/questions?

Thanks!