# Functional pearl: zero-knowledge testing for module interfaces

Thomas Braibant Jacques-Henri Jourdan Jonathan Protzenko Gabriel Scherer
INRIA
http://gallium.inria.fr/blog/

## Abstract

In spite of recent advances in full program certification, testing remains a widely-used component of the software development cycle. Various flavors of testing exist: popular ones include *unit testing*, which consists in manually crafting test cases for specific parts of the code base, as well as *quickcheck-style* testing, where instances of a type are automatically generated to serve as test inputs.

These classical methods of testing can be thought of as *internal* testing: the test routines access the internal representation of whatever module should be checked. We propose a new method of *external* testing where test code checks an *abstract* data structure. Our new testing method takes a description of a *module signature*, then builds sequences of function calls that generate elements of the abstract type just like any other client code. Counter-examples, if any, are then presented to the user.

***Categories and Subject Descriptors*** CR-number [*subcategory*]: third-level

***Keywords*** functional programming, testing, quickcheck

## 1. Introduction

Software development is hard. Industry practices still rely, for the better part, on tests to ensure the functional correctness of programs. Even in more sophisticated circles, such as the programming language research community, not everyone has switched to writing all their programs in Coq. Testing is thus a cornerstone of the development cycle. Moreover, even if the end goal is to fully certify a program using a proof assistant, it is still worthwhile to eliminate bugs early by running a cheap, efficient test framework.

Testing boils down to two different processes: generating test cases for test suites; and then verifying that user-written assertions and specifications of program parts are not falsified by the test suites.

QuickCheck is a popular, efficient tool for that purpose. First, it provides a combinator library based on type-classes to build test case generators. Second, it provides a principled way for the users to specify properties over functions. For instance, users may write predicates such as "reverse is an involution". Then, the QuickCheck framework is able to create *instances* of the type being tested, e.g., lists of integers. The predicate is tested against these test cases, and any counter-example is reported to the user.

Our novel approach is motivated by some limitations of the QuickCheck framework. When users create trees, for instance, not only do they have to specify that leaves should be generated more often than nodes (for otherwise the tree generation would not terminate), but they also have to rely on a global size measure to stop generating new nodes after a while. It is thus up to the user of the library to implement their own logic for generating the right instances, within a reasonable size limit, combining the various base cases.

We argue that these low-level manipulations should be taken care of by the library. When generating binary search tree instances, one ends up re-implementing a series of random additions and deletions, which are precisely the function that the code to be tested for exports. What if the testing framework could, by itself, combine functions exported by the module we wish to test, in order to build instances of the desired type? As long as the the module exports a correctness predicate, all the testing library needs is functions that *return* $t$'s.

In the present document, we describe a library that does precisely that, dubbed ArtiCheck. The library is written in OCaml. While QuickCheck uses a combination internal testing and type classes, our library performs external testing and relies on GADTs.

## 2. The essence of external testing

In the following section, we illustrate the core principles of external testing by taking a small example. Reasoning on that small example, we try to design a very basic library that performs external testing. The library has several shortcomings and fails to address several salient points, but illustrates well our purpose. Properly designing such a library is the topic of §3.

Our running example in this section is the trivial module which exports an abstract type `t` enforcing the invariant that elements of type `t` describe sorted integer lists.

```
module type SIList = sig
  type t

  val empty: t
  val add: t -> int -> t
  val check: t -> bool
end
```

The `check` function dynamically checks that an element of type `t` satisfies the internal invariant. The module admits a straightforward implementation, as follows.

```
module SIList = struct
  type t = int list

  let empty = []
```



*ICFP '14*, September 1–3, 2014, Copenhagen, Denmark.

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
  let rec add x = function
    | [] -> [x]
    | t::q -> if t<x then t::add x q else x::t::q

  let rec check = function
    | [] | [_] -> true
    | t1::(t2::_ as q) -> t1 <= t2 && check q
end
```

In order to keep track of the set of `t`'s we have built so far, we need what we call a *type descriptor* for `t`.

```
type 'a ty = {
  (* other implementation details omitted *)
  mutable enum: 'a list;
  fresh: ('a list -> 'a) option;
}
```

A type descriptor `'a ty` keeps track of all the *instances* we have created so far via its `enum` field. In the case of built-in types or, phrased differently, types that are *external* to the library, there is no point in constructing them manually. For types such as `int`, we therefore provide a `fresh` function that generates a fresh integer different from all we have generated so far.

The goal of ArtiCheck is to play the role of the client code. In hindsight, ArtiCheck behaves like a robot that imitates a client of the library. Things that the client are allowed to do with the above signature are: manipulating the `empty` element and calling `add` to generate new elements of type `t`. We want ArtiCheck to figure this out and generate sequences of calls to `add`.

In essence, we want to represent well-typed applications in the simply-typed lambda-calculus. This can be embedded in OCaml using GADTs. We define the GADT `('f, 'r) fn`. The type describes ways of producing instances of type `'r` using a function of type `'f`. We call it a *function descriptor*.

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

(* Helpers for creating [fn]'s. *)
let (@->) ty fd = Fun (ty,fd)
let returning ty = Constant ty
```

The `Ret` case describes a constant value, which has type `'a` and produce one instance of type `'a`. For reasons that will soon become apparent, we also record the descriptor of type `'a`. `Fun` describes the case of a function from `'a` to `'b`: using the descriptor of type `'a`, we can apply the function to obtain instances of type `'b`; combining that with the other `('b, 'c) fn` gives us a way to produce elements of type `'c`, hence then `('a -> 'b, 'c) fn` conclusion.

```
let (>>=) li f = List.flatten (List.map f li)

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

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

The `eval` function is central: taking a function descriptor `fd`, it recurses over it, thus refining the type of its argument `f`. The use of GADTs allows us to statically prove that the `eval` function only ever produces instances of type `b`. The `codom` function allows one to find the type descriptor associated to the return value (the codomain) of an `fn`.

Using the two functions above, it then becomes trivial to generate new instances of `'b`.

```
let use (fd: ('a, 'b) fn) (f: 'a) =
  let prod = eval fd f in
  let ty = codom fd in
  List.iter (fun x ->
    if mem x ty then () else ty.enum <- x::ty.enum
  ) prod
```

The function takes a function descriptor along with a matching function. The `prod` variable contains all the instances of `'b` we just managed to create; `ty` is the descriptor of `'b`. We store the new instances of `'b` in the corresponding type descriptor.

In order to wrap this up nicely, one can define *signature descriptors*. An entry in a signature descriptor is merely a function that produces a certain type `'a` along with its corresponding function descriptor. Once this is done, the user can finally call our library and test the functions found in the signature description.

```
type sig_elem = Elem : ('a,'b) fn * 'a -> elem
type sig_descr = (string * sig_elem) list
let si_t  =
  (* create a descriptor for [SIList.t]... *)
let int_t =
  (* ...and one for [int], with a [fresh] function *)

let sig_of_silist = [
  ("empty", (returning si_t, SIList.empty));
  ("add", (int_t @-> si_t @-> returning si_t, SIList.add));
]

let _ =
  Arti.check sig_of_silist SIList.check
```

The `Arti.check` function repeatedly calls `use` on the items found in the signature, until the desired number of instances have been created. We call `use` for each function in the signature several times: failing that, the only applications we could ever build would be of the form `add empty n`. We then obtain the descriptor for `SIList.t` and check that each instance satisfies the `SIList.check` predicate.

## 3. Describing a type

In the context of our testing framework, we wish to describe a type using a set of *instances*, that is, of inhabitants of the type. The following data structure makes up what we call a *type descriptor*.

```
type 'a ty = {
  (* other implementation details omitted *)
  mutable enum: 'a PSet.t;
  fresh: ('a PSet.t -> 'a) option;
}
```

Type descriptors record a persistent set of instances via the `enum` field. Descriptors for ground types, such as `int`, are equipped with a generator of `fresh` instances that are guaranteed to be different from the others.

## 4. Bits

Bernardy et al. [1] describe a systematic way of reducing the testing of polymorphic functions to the testing of specific monomorphic instances of these functions. Given a polymorphic property,

the correctness of the reduced (monomorphic) property entails the correctness of all other instanciations. This yields a significant reduction in the necessary test cases. They informally argue that their technique is efficient compared to the standard praxis of substituting `int` for polymorphic types. Note however that both solutions to the problem of testing polymorphic functions must be applied at the meta-level. That is, the user has to pick the right instanciation of polymorphic type variables; this cannot be done automatically inside the host language.

## References

[1] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer, 2010.