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 present section, we illustrate the essential idea of external testing through a simple example, which is that of a module `SIList` whose type `t` represents sorted integer lists. The invariant is maintained by making `t` abstract and requiring the user to go through the exported functions `empty` and `add`.

This section, unfolding from the initial example, introduces the key ideas of external testing: a GADT type that describes well-typed applications in the simply-typed lambda calculus; a description of module signatures that we wish to test; type descriptors that record all the instances of a type that we managed to construct.

The point of view adopted in this section is intentedly simplistic. The design, as presented here, contains several obvious shortcomings. It allows, nonetheless, for a high-level overview of our principles, and paves the way for a more thorough discussion of our design which appears in §3.

Here is the signature for our module of sorted integer lists.

```
module type SIList = sig
  type t

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


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

The `check` function represents the *invariant* that the module pretends it preserves. The module admits a straightforward implementation, as follows.

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

  let empty = []

  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
```

Roughly speaking, our goal is to generate, as if we were *client code* of the module, instances of type `t` using only the functions exported by the module. Therefore, one of our first requirements is a data structure for keeping track of the `t`'s created so far. We also need to keep track of the integers we have generated so far, since they are necessary to call the `add` function: ArtiCheck will thus manipulate several `ty`'s for all the types it handles.

```
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 in its `enum` field. Built-in types such as `int` do not belong to the set of types whose invariants we wish to check. For such types, we provide a `fresh` function that generates an instance different from all that we have generated so far.

From the point of view of the client code, all we can do is combine `add` and `empty` to generate new instances. ArtiCheck, as a fake client, should thus behave similarly and automatically perform repeated applications of `add` so as to generate new instances. We thus need a description of what combinations of functions are legal for ArtiCheck to perform.

In essence, we want to represent well-typed applications in the simply-typed lambda-calculus. This can be embedded in OCaml using generalized algebraic data types (GADTs). We define the GADT `('f, 'r) fn`, which describes ways to generate 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 the `('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 of 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. The library calls `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`. The library then fetches the descriptor for `SIList.t` and check that each instance satisfies the `SIList.check` predicate.

3. Implementing ArtiCheck

The simplistic design we introduced in §2 conveys the main ideas behind ArtiCheck, yet fails to address a wide variety of cases. The present section reviews the issues with the current design and incrementally addresses them.

3.1 A better algebra of types

The simply-typed lambda calculus that we introduced only contains constants and functions. While one can theoretically encode sums and products using functions, it seems reasonable to have a built-in notion of sums and products in our language.

One of the authors naïvely suggested that the data type be extended with cases for products and sums, such as:

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

It turns out that the branch above does not describe products. If `'a` is `int -> int` and `'b` is `int -> float`, not only do the `'c` parameters fail to match, but the `'a * 'b` parameter in the conclusion represents a pair of functions, rather than a function that returns a pair! Another snag is that the type of `eval` makes no sense in the case of a product. If the first parameter of type `('a, 'b) fn` represents a way to obtain a `'b` from the product type `'a`, then what use is the second parameter of `eval`?

In light of these limitations, we take inspiration from the literature on focusing and break the `fn` type into two distinct GADTs.

- The GADT `('a, 'b) negative` (`neg` for short) represents a *computation* of type `'a` that produces a result of type `'b`.
- The GADT `'a positive` (`pos` for short) represents a *value*, that is, the result of a computation.

```
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
```

The `pos` type represents first-order data types: products, sums and atomic types, that is, whatever is on the rightmost side of an arrow. We provide an injection from positive to negative types: a value of type `'a` is also a constant computation.

We do *not* provide an injection from negative types to positive types: this would allow nested arrows, that is, higher-order types. One can take the example of the `map` function, which has type `('a -> 'b) -> 'a list -> 'b list`: we explicitly disallow representing the `'a -> 'b` part as a `Fun` constructor, as it would require us to synthesize instances of a function type. Rather, we ask the user to represent `'a -> 'b` as a `Ty` constructor; in other words, we ask the user to supply their own test functions as if they were a built-in type.

Since our GADT does not accurately model tagged, n-ary sums of OCaml, we provide a last `Bij` case that allows the user to provide a two-way mapping between a built-in type (say, `'a option`) and its ArtiCheck representation (`() + 'a`). That way, ArtiCheck can work with regular OCaml data types by converting them back-and-forth.

This change of representation incurs some changes on our evaluation functions as well. The `eval` function is split into several parts, which we detail right below.

```
let rec apply: type a b. (a, b) neg -> a -> b list =
  fun ty v -> match ty with
  | Fun (p, n) ->
      produce p |> concat_map (fun a -> apply n (v a))
  ...
and produce: type a. a pos -> a list =
  fun ty -> match ty with
  | Ty ty -> Ty.elements ty
  | Prod (pa, pb) ->
      cartesian_product (produce pa) (produce pb)
  ...
let rec destruct: type a. a pos -> a -> unit =
  function
  | Ty ty -> (fun v ->
      remember v ty)
  | Prod (ta, tb) -> (fun (a, b) ->
      destruct ta a;
      destruct tb b)
  ...

(* Putting it all together *)
let _ =
  ...
  let li = apply fd f in
  List.iter (destruct head) li;
  ...
```

Let us first turn to the case of *values*. In order to understand what ArtiCheck ought to do, one may ask themselves what the user can do with values. The user may destruct them: given a pair of type `'a * 'b`, the user may keep just the first element, thus obtaining a new `'a`. The same goes for sums. We thus provide a `destruct` function, which breaks down positives types by pattern-matching, populating the descriptions of the various types it encounters as it goes.

Keeping this in mind, we must realize that if we can use a positive type `'a` to obtain a `'b` (`apply`), the user may use any possible means to produce an `'a`: if `'a` is a product, they will use every possible combination of elements that are available to them; if `'a` is a sum, they will either every choice from the type of either. We must therefore devise a function `produce` that represents the entire set of possible choices for a positive type.

The `apply` function, just like before, takes a *computation* along with a matching description, and generates a set of `b`. However, it now relies on `product` to exhibit all possible instances of a type before passing these instances on to the actual function.

We are now able to accurately model a calculus rich enough to test realistic signatures involving records, option types, and various ways to create functions.

3.2 Efficient representation of a set of instances

The (assuredly naïve) scenario above reveals several pain points with the current design.

- We represent our sets using lists. We could be more efficient.
- If some function takes, say, a tuple, the code as it stands will construct the set of all possible tuples, perform a `map`, then finally call `destruct` on each resulting element to collect instances. Naturally, memory explosion ensues. We propose a symbolic algebra for *sets of instances* that *mirrors* the structure of positive types and avoids the need for holding all possible combinations in memory at the same time.
- Take the example of logical formulas. The naïve strategy consists in generating formulas with one combinator in a first round, then passing the freshly created instances into a second round to generate instances with two combinators. One may wish to be smarter and pass the instances generated by, say, `mk_and` to `mk_xor` so as to generate formulas with two opera-

tors in one round. This approach is not feasible, because we run into non-termination issues and fairness problems.

- Interesting instances containing three combinators are only reached after we've exhausted all possible instances with two or less combinators. This breadth-first search of the instance space is sub-optimal. Can we do better?

Sets of instances The first, natural optimization that comes to mind consists in dropping lists in favor of a more sophisticated data type. We replace lists with a module `PSet` of persistent sets, implemented as red-black trees.

Not holding sets in memory A big source of inefficiency is the call to the `cartesian_product` function above (§3.1). We hold in memory at the same time all possible products, then pipe them into the function calls so as to generate an even bigger set of elements. Only when the set of all elements has been constructed do we actually run `destruct`, only to extract the instances that we have created in the process.

Holding in memory the set of all possible products is too expensive. We adopt instead a *symbolic representation of sets*, where unions and products are explicitly represented using constructors. This mirrors our algebra of positive types.

```
type _ set =
  | Set    : 'a PSet.t -> 'a set
  | Bij    : 'a set * ('a, 'b) bijection -> 'b set
  | Union   : 'a set * 'b set -> ('a,'b) sum set
  | Product : 'a set * 'b set -> ('a * 'b) set
```

This does not suppress the combinatorial explosion. The instance space is still exponentially large; what we gained by changing our representation is that we no longer hold all the "intermediary" instances in memory *simultaneously*. This allows us to write an `iter` function that constructs the various instances on-the-fly.

```
let rec iter: type a. (a -> unit) -> a set -> unit =
fun f s -> match s with
  | Set ps ->
     PSet.iter f ps
  | Union (pa,pb) ->
     iter (fun a -> f (L a)) pa;
     iter (fun b -> f (R b)) pb;
  | Product (pa,pb) ->
     iter (fun a -> iter (fun b -> f (a,b)) pb) pa
  | (* ... *)
```

Piping and non-termination In order to push the optimization above further, one can choose to perform the call to `remember` directly inside the `Ret` case of `apply`. That way, `apply` could just fill in the type descriptors using the global, mutable state and return unit, thus avoiding the need for intermediary lists of instances. Also, calling `remember` directly eliminates the need to store duplicate items, as the function automatically takes care of dropping an instance if we are already aware of it.

This seemingly innocuous optimization raised non-termination issues. We explain why, in the hope that it serves as an example for future generations ("kids, don't do mutable state").

Consider the case of a function that has type `t -> t -> t`. The outer call to `apply` binds the list of instances of `t` via `let l = ty.enum`. For each element of `l`, a recursive call to `apply` takes place (for the inner `t -> t` function), which looks up the current value of `ty.enum`. Since each inner call populates `ty.enum` itself, for each new recursive call of `apply`, the value of `ty.enum` grows bigger and bigger. The programs terminates by exhausting its memory space without even returning from the outer call to `apply`.

We solved this by taking a snapshot of our negative types before calling `apply`. That way, we save a copy of the arguments that are to be applied in each `Fun` case.

Fairness of our search space Snapshotting enforces a breadth-first search of the instance space. The initial set of instances is fed through the available functions, and we iterate the process, until we've obtained a satisfactory number of instances for each one of the types we wish to test.

The distribution of instances is skewed: there are more instances obtained after `n` calls than there are after `n+1` calls. It may thus be the case that by the time we reach three of four consecutive function calls, we've hit the maximum limit of instances allowed for the type, since it often is the case that the number of instances grow exponentially.

We plan to implement a random search of the instance space and tweak our exploration procedures so that "interesting" instances pop up early.

3.3 Instance generation as a fixed point computation

A natural framework for expressing instance generation is a system of equations. Equations between variables (type descriptors) describe ways of obtaining new instances (by applying functions to other type descriptors). All we need is an upper bound on the desired number of instances for each variable, to make sure that this is actually a fixed-point computation.

```
module Fix = sig
  type      eqns = var -> rhs
  and        rhs = valuation -> property
  and valuation = var -> property
end
```

The signature above exposes the essence of `Fix`, the fixed-point computation framework we use. A system of equations maps a variable to a right-hand side. Each right-hand side can be evaluated by providing a valuation so as to obtain a property. Valuations map variables to properties.

A perhaps tempting way to fit in this setting would be to define variables to be our `'a ty` (type descriptor) and properties to be `'a lists` (the instances we have built so far). This doesn't work as is: since there will be multiple values of `'a` (we generate instances of different types simultaneously), type mismatches are to be expected. One could, after all, use yet another GADT and hide the `'a` type parameter behind an existential variable.

```
type _ var = Atom: 'a ty
type _ property = Props: 'a list
```

The problem is that there is no way to statically prove that having an `'a var` named `x`, calling `valuation x` yields an `'a property` with a matching type parameter. One could rewrite the `Fix` module to parameterize the `var` and `property` types over a type variable `'a`, but then one runs into higher-rank polymorphism which is not trivial to express in OCaml. Fortunately, we can use a trick that involves mutable state.

Recall that the definition of `'a ty` is a record with an `enum` field that holds all the instances generated so far. Storing instances as properties is actually redundant: what we do instead is store a pair of integers.

```
type _ var = Atom: 'a ty
type property = int * int
```

The first integer stands for the maximum number of instances we wish to generate; the second integer stands for the number of instances we have generated *so far*. Every time a right-hand side is evaluated, we generate new instances using the functions at hand;

we update the second integer and mutate the `enum` field of our variable, which fortunately is passed to us as the first parameter.

4. Examples

5. 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.