ArtiCheck: well-typed generic fuzzing 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 data structure should be checked. We propose a new method of *external* testing where the library only deals with a *module interface*. The data structures are exported as *abstract types*; the testing framework behaves just like regular client code and combines functions exported by the module to build new elements of the various types. 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* of `'a` 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 = Ret 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
    | Ret ty -> ty
    | Fun (_,fd) -> codom fd
```

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 problems. 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 via the `Ret` constructor: 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.

Our GADT does not accurately model tagged, n-ary sums of OCaml, nor records with named fields. We thus add a last `Bij` case; it 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.enum
  | 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 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. (The `remember` function records all instances we haven't encountered yet in the type descriptor `ty`.)

Keeping this in mind, we must realize that if a function takes an `'a`, the user may use any `'a` it can produce to call the function. For instance, in the case that `'a` is a product type `'a1 * 'a2`, then *any* pair of `'a1` and `'a2` may work. We introduce a function called `produce`, which reflects the fact the user may choose any possible pair: the function exhibits the entire set of instances we can build for a given type.

Finally, 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 exhaustively exhibit all possible arguments one can pass to the 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 use a more efficient data structure.
- If some function takes, say, a tuple, the code as it stands will construct the set of all possible tuples, `map` the function over the set, 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.
- A seemingly trivial optimization sends us off the rails by generating an insane number of instances. We explain how to optimize further the code while still retaining a well-behaved generation.
- Fairness issues arise. Take the example of logical formulas. One may try to be smart: starting with constants, one may apply `mk_and`, then pass the freshly generated instances to `mk_xor`.

A consequence is that all the formulas with two combinators start with `xor`. If we just keep an iterative process and do not chain the instance generation process, formulas 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 polymorphic, 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 combinatorial explosion 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` and a corresponding type descriptor for `t` named `ty`. 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`. No copy is involved: function arguments (positive types) are represented in memory as persistent, pure symbolic sets. That way, we keep 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

The `apply`/`destruct` combination only demonstrates how to generate new instances from one specific element of the signature. We need to iterate this recipe on the whole signature, feeding the new values obtained to the operators that consume them as input.

This part of the problem naturally presents itself as a fixpoint computation, defined by a system of equations. Equations between variables (type descriptors) describe ways of obtaining new instances (by applying functions to other type descriptors). Of course, to ensure termination we need to put a bound on the number of generated instances. This is a classic in algorithms presented as fixpoints problems, where the lattice space is artificially made finite to get termination.

Implementing an efficient fixpoint computation is a *surprisingly interesting* activity, and we are happy to use an off-the-shelf fixpoint library, François Pottier's `Fix`, to perform the work for us. `Fix` can be summarized by the signature below, obtained from user-defined instantiations of the types `variable` and `property`.

```
module Fix = sig
  type valuation = variable -> property
  type rhs = valuation -> property
  type equations = variable -> rhs

  val lfp: equations -> valuation
end
```

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. Solving a system of equations amounts to calling the `lfp` function which, given a set of equations, returns the corresponding valuation.

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); the equations derived from any signature would then describe ways of obtaining new instances by applying any function of the signature. 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 variable = Atom: 'a ty -> variable
  type property = Props: 'a set -> property
```

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. This is precisely where the mutable state in the `'a ty` type comes handy: even it is only passed as input to the system of equations, we can also use it to store the output; the `property` type needs not mention the type variable `'a` anymore, which removes any typing difficulty – or need to change `Fix`'s interface.

We still need the `property` type to be a rich enough lattice to let `Fix` decides when to stop iterating: it should come with equality- and maximality-checking functions, used by `Fix` to detect that the fixpoint is reached. The solution is to define `property` as the number of instances generated so far (and the bound we have chosen in advance):

```
type variable = Atom : 'a ty -> variable
type property = { required : int; produced : int }
let equal p1 p2 = p1.produced = p2.produced
let is_maximal p = p.produced >= p.required
```

4. Expressing correctness properties

The `SIList` example uses `check` function that runs on top of the simplest possible testing function, called `counter_example`:

```
val counter_example :
  'a positive -> ('a -> bool) -> 'a option
```

This functions takes the description of some (positive) datatype `'a`, iterates on the produced values at this type, and checks that a predicate `'a -> bool` holds, or returns a counter-example. At a more abstract level, this means that we are checking a property of the form

$$\forall (x \in t), T(x)$$

where T is simply a boolean expression. Using product types allows to simulate multiple quantifiers; for example, the typical formula of association maps

$$\forall (m \in \mathtt{map}(K, V))\ \forall (k \in K)\ \forall (v \in V),\\ \mathtt{lookup}(\mathtt{k}, \mathtt{insert}(\mathtt{k}, \mathtt{v}, \mathtt{m})) = \mathtt{v}$$

can be expressed as

```
let lookup_insert_prop (k, v, m) =
  lookup k (insert k v m) = v
let () = assert (None =
  let kvm_t = k_t *@ v_t *@ map_t in
  counter_example kvm_t lookup_insert_prop)
```

It is then natural to wonder what is a good language to describe the correctness properties we want to check. We have instictively used first-order logic, and are able to express formulas with prenex universal quantifiers followed by a quantifer-free formula; the ability to generate random elements gives a "test semantics" to (prenex) universal quantifiers. Can we do better? In particular, can we capture the full language of first-order logic, as a reasonable description language for tests in a practical framework?

There are various reasons why it is difficult to support full first-order logic as a specification language for tests using only random generation – as opposed to more structured verification approaches such as SMT solvers or finite model finders [**?**]. For example, it is awkward to give a test semantics to an existential formula $\exists (x \in t).T(x)$. The user expects a test to tell you little if it returns "yes", but always have found a bug (with an exercizing input) if it returns "no". If you generate a bounded number of elements of t randomly, the fact that none satisfy T may not indicate a bug, simply that you haven't tested the good elements. Trying to distinguish absolute (positive or negative) results from probabilistic results opens a world of complexity that we chose not to explore.

Surprisingly to us, there does not seem to be a consensus in the random testing literature on an expressive, well-defined subset of first-oder logic. The simplest subset one directly thinks of is the formulas of the form

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

where $P(x_1, \dots, x_n)$ (the *precondition*) and $T(x_1, \dots, x_n)$ (the *test*) are both quantifier-free formulas. The reason to give a specific status to this implication is to count differently tests that succeeded because the precondition was not verified, which often bring little confidence and should not be counted as successes.

5. Examples

5.1 Red-black trees

The (abridged) interface exported by red-black trees is as follows. The module provides iteration facilities over the tree structure through the use of *zippers*. Our data structures are persistent.

```
module type RBT = sig
  type 'a t

  val empty : 'a t
  val insert : 'a -> 'a t -> 'a t

  type direction = Left | Right
  (* type 'a zipper *)
  type 'a ptr (* = 'a t * 'a zipper *)

  val zip_open : 'a t -> 'a ptr
  val zip_close : 'a ptr -> 'a t

  val move_up : 'a ptr -> 'a ptr option
  val move : direction -> 'a ptr -> 'a ptr option
end
```

This examples highlights several strengths of ArtiCheck.

First, two different types are involved: the type of trees and the type of zippers. While an aficionado of internal testing may use the `empty` and `insert` functions repeatedly to create new instances of `'a t`, it becomes harder to type-check calls to *either* `insert` or `zip_open`. Our framework, thanks to GADTs, generates instances of both types painlessly and automatically.

Second, we argue that a potential mistake is detected trivially by ArtiCheck, while it may turn out to be harder to detect using internal testing. If one removes the comments, the signature reveals that pointers into a tree are made up of a zipper along with a tree itself. It seems fairly natural that the developer would want to reveal the `zipper` type; it is, after all, a fundamental feature of the module. An undercaffeinated developer, when writing internal test functions, would probably perform sequences of calls to the various functions. What they would fail to do, however, is destructing pairs so as to produce a zipper associated with *the wrong tree*. This particularly wicked usage would probably be overlooked. ArtiCheck successfully destructs the pair and performs recombinations, to finally output:

```
TODO: fix the code so that it terminates
  ... and put the error message here
```

5.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) represent trees for deciding logical formulas. The defining characteristic of BDDs is that they enforce *maximal sharing*: wherever two structurally equal subformulas appear, they are guaranteed to refer to the same object in memory. A consequence is that performing large numbers of function calls does not necessarily means using substantially more memory: it may very well be the case that significant sharing occurs.

We mentioned earlier that our strategy for external testing amounted, in essence, to representing series of well-typed function calls in the simply typed lambda calculus using in GADT. If we only did that and skipped section §3, externally-testing BDDs would be infeasible, as we would end up representing a huge number of function calls in memory.

Conversely, with the design we exposed earlier, we merely record new instances as they appear without holding the entire set of potential function calls in memory. This allows for an efficient, non-redundant generation of test cases (instances).

5.3 AVL trees

AVL trees are a classic of programming interviews; many a graduate student has been scared by the mere mention of them. It turns out that tenured professors *should* be scared too: the OCaml implementation of maps, written using AVL trees by a respectable researcher, contained a bug that went unnoticed for more than ten years. The bug was discovered when another enthusiastic researcher set out to formalize the said library in Coq. The bug was fixed, and all was well. Out of curiosity, we decided to run ArtiCheck on the faulty version of the library. After registering only four functions with ArtiCheck, the bug was correctly identified by our library, with arguably less pain than the full Coq formalization required.

6. Conclusion

We have presented the design of ArtiCheck, a novel library that allows one to check the invariants of a module signature by simulating user interaction with the module. ArtiCheck behaves like a fake client: it calls functions, constructs and destructs products or sums, and for each element check that the invariants are verified. The key to performing this in a generic, abstract manner relies on GADTs, which abstract the different types that may be manipulated into a common representation.

We identified various performance problems that arise. The library handles them via a symbolic representation of types in combination with a little bit of mutable state to avoid handling large, intermediary results in memory.

The result is a self-contained library that wraps the core concepts of *external testing* and offers clients a cheap and efficient way to test their programs. The library, for instance, successfully detects infamous issues such as the AVL re-balancing issue in the standard library of OCaml, with a much lower cost than a complete Coq proof of the module.

While the library exposes the essence of *external testing* and has already proven worthwhile, we believe there is potential for improvement and expansion into a fully-fledged testing library.

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.