

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.

Quickcheck is a popular, efficient tool for that purpose. Users write predicates for the correctness of their elements, such as “applying reverse twice yields the same list”. Using type classes, the quickcheck framework is then able to create *instances* of the type being tested, which are then tested against the predicate. Any counter-examples are 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 ter-

minate), 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 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 “artichek”. 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. Bits

Bernardy et al. [?] 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 instantiations. 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 instantiation of polymorphic type variables; this cannot be done automatically inside the host language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-NNNN-NNNN-N/YY/MM...\$15.00.

<http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn>