Why Testing Matters in Functional Programming
Position Paper

Manfred Widera

Department of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany,
Manfred.Widera@fernuni-hagen.de

Abstract

The testing of programs is an approach that has been neglected by a part of the users and researchers in the area of functional programming for some time. Lately, it has received a little more attention. In this paper we compare program testing with other approaches to software quality, and we motivate why testing is important even when programming in a high level paradigm like functional programming. We discuss which parts of the testing process are affected by the chosen programming paradigm and present areas in testing functional programs that are worth some further investigation.

1 MOTIVATION

The functional programming paradigm allows the implementation of programs on a high abstraction level. The risk of programming errors is reduced by this abstraction: error-prone tasks like memory management are transfered from the programmer to the language, and high level constructs for composing individual modules, like higher order functions and lazy evaluation [Hug89], help to reduce the complexity of software systems.

The natural error resistance of functional programming languages led to a high degree of confidence into the correctness of functional programs. For some time, programmers and researchers expected to reach the correctness of functional programs by the (more or less) exclusive application of more formal methods than testing, e.g. powerful type systems and verification approaches. Testing of functional programs was, therefore, not considered.

Indeed, many of the programming errors occurring in functional programs can be detected by employing these tools and, especially, by using static type checkers. A smaller number of errors does, however, not affect the type correctness and can be quite hard to detect. (E.g. in the usual programming languages there is no type for a sorted list of some type A. Hence, when assuming a list to be ordered and violating this property at a distant code section, this error cannot be detected by the usual type systems.) Furthermore, formal verification tools need an already correct, formal specification of the expected program properties. They cannot detect errors that occur in both, this specification and the program in a consistent form.

Up to now, a small number of approaches on testing functional languages exists [CH00, CH02], [KATP03, KP05], [Wid04a, Wid05b]. Testing has, however,

not reached the full attention it deserves, so far. The aim of this paper is to motivate why testing issues should become a trend in functional programming. We give an overview over the issues that should be addressed when considering the testing of functional programs. Based on the special properties of functional programming languages, there are special chances, but also special pitfalls in the testing of functional programs. We identify these issues, discuss how they are addressed by existing publications and describe useful directions of future research in the area of testing functional programs.

In Sec. 2 we discuss the properties of different approaches on improving software correctness, and motivate why testing is still an important part of the program development process, even when using a high level programming paradigm like functional programming. Section 3 identifies the testing steps that are influenced by the choice of the programming paradigm. We identify the choice of appropriate test data and the judgement whether a computed test result is correct as the main problems arising in testing functional programs. The choice of appropriate test inputs is addressed in Sec. 4, focusing on the common problems and differences of manual and automatic test case generation. In Sec. 5 we analyze the judgement of test results. Here we identify the potential of functional programs to return function values and infinite lazy structures as the main problem for both, manual result inspection and the implementation of test predicates. Section 6 finally contains our conclusions.

2 WHY FUNCTIONAL PROGRAMS NEED TO BE TESTED

Testing is not the main concern of many users and researchers in the area of functional programming. Some of them see a discrepancy between the high abstraction level used in functional programs and the low level testing approach.

The aim of this paper is to overcome this aversion of functional programmers against testing and to show that the testing of functional programs is a worthwhile area of research.

In this section, testing of functional programs is analyzed from several different points of view, and the need for testing in the functional programming context is motivated.

2.1 The Aim of Software Testing

Myers [Mye79] describes the aim of software testing as follows.

> Testing is the process of executing a program with the intent of finding errors.

From this quote we can especially conclude what testing does *not* try to do.

- In contrast to verification techniques, testing does not aim at proving the correctness of a program.

- Testing does not try to identify erroneous pieces of code. This is done using tracing and debugging approaches (e.g. [Gil00], [Nai97], [Chi01, WCBR01]).

Conversely, the mentioned approaches are of little use in the main domain of testing, i.e. for detecting errors in a software component.

We conclude that testing, tracing and verification approaches are independent and cannot replace each other. Testing should be applied early to an implemented software component/software system. It can be followed by tracing and debugging approaches for the error correction, and by verification approaches when the correctness of the software is assumed.

2.2 Levels of Formalization

Programming is a layered task of specification and formalization of program properties. The following layers of formalization exist (where the intermediate stages need not occur in each program development process).

1. An intuitive model of the application scenario and the needed properties of the new software is developed.

2. The intuitive model is fixed in a textual specification.

3. One or several levels of formal specifications of the program properties are generated, covering among others the input/output behavior of the program and the definition of modules and interfaces.

4. The program is implemented based on the most special specification of its properties.

On each of these levels the program properties are formalized and specified to a higher degree. With a higher degree of formalization the information is better suited for the application of formal methods but less understandable by humans.

In different programming methodologies these levels may not be reached in an sequential order. E.g. extreme programming [XP06] makes heavy use of iterated tasks. In every iteration step, however, the programmer needs a model of the implementation goal before he starts the implementation, and he may wish to have a more formal specification, as well.

The common idea of many testing and verification approaches is to compare the properties of the implementation against the properties stated in one of the previous stages. The approaches differ in the level of the reference stage to use.

Formal verification and proof methods need a formal specification to act against (i.e. the formal tools are based on some output of Stage (3)). In contrast, testing can use the informal specifications written down in Stage (2). For the final testing stages, that are done by the customer, even the informal model of Stage (1) acts as basis for the judgement of the testing results.

Errors can occur in each stage of specification. These errors usually cumulate and an error in one stage usually causes all following stages to contain the corresponding error. From this observation we can state that testing cannot be replaced by more formal methods because it is capable of discovering errors in more of the specification steps than this is possible for the formal tools.

Furthermore, programming methodologies like extreme programming and rapid prototyping aim at reducing the amount of formalized specifications generated during the development of a prototype or software system. In such cases, the existing specifications do not suffice for a verification approach. In the absence of a formal specification, a good testing policy is important to reach a sufficient level of software quality.

2.3 Validation vs Verification

Distinguishing the notions of *verification* and *validation* can be of further help to understand the importance of testing. According to Rakitin [Rak97, p. 129] these approaches aim at answering the following questions during the software development process.

- *Verification*: “Are we building the product right?”
- *Validation*: “Did we build the right product?”

Besides the search for errors in the software product, testing can be used for validation purposes. Especially, this applies to the final testing stages focusing on the completed software product. Here, the late testing stages can show whether the completed software fits as expected into the intended application environment.

In general, verification and validation are such different goals, that the usual verification approaches cannot be used to perform validation as well. In contrast, testing is a powerful tool for the validation. For this reason, software projects making use of functional programming should not abandon testing.

3 CHALLENGES IN TESTING FUNCTIONAL PROGRAMS

The testing process during the software development is usually divided into several stages [Rak97].

1. The *unit* or *module testing* stage is applied to individual units of the software product early in the software development process. It is used to detect errors in the logic of the components.

2. The *integration testing* stage applies to a number of integrated units or modules. Its focus is on the uniform interpretation and implementation of the interfaces by the individual components.

3. The *system testing* stage focuses on the completed software product. It is sometimes divided into *validation testing* (test that all of the fixed specifications are met by the software) and *acceptance testing* (test that the software meets the customer requirements).

The dependencies of the testing process on the programming paradigm decrease when proceeding to later stages of testing. For the acceptance tests, the programming languages used for the tested software product do not matter at all: the customer is just interested in the question whether the software product meets all requirements. He usually does not want to consider the underlying programming concepts. In unit testing one has, however, to cope with different details of testing that are programming language dependent.

Two main issues arise in unit testing (and also partly in integration testing) that must be considered with functional programming languages in mind.

- The choice and generation of appropriate test cases.
- The determination whether the generated test outputs are correct.

For the choice of the appropriate test cases one can distinguish black box approaches not considering the code under testing and white box approaches based on the structure of the tested code. Especially the white box approaches which are usually used during the unit testing stage must fit the programming paradigm of the tested program because of the focus on its structure.

The generation of test cases can be done automatically from the type specification of the functions under testing, especially if the input domain of the function can be described precisely by the underlying type system.

In checking the computation results, special problems arise because of the possible complexity of values in functional languages. For infinite, lazy data structures the inspection can just be partial, and for function type result values the check for correctness must be customized for every individual result function.

For the two main issues, the choice/generation of test sets and the correctness checking of the results, the following two sections discuss the current situation in the existing testing approaches and address areas of future investigations.

4 THE CHOICE OF TEST DATA

4.1 Required Properties of Test Sets

Traditionally, test data are selected or judged according to a number of well-known principles [Lig02]. These principles can be divided into two main groups.

- *Black box testing* chooses test sets without considering the tested code. It is just based on the specifications of the intended I/O-behavior.
- *White box testing* determines the appropriateness of a test set based on the structure of the tested code fragment.

4.1.1 Black Box Testing

In *black box testing*, test cases are chosen without considering the code of the program part under testing. This approach chooses the test cases solely based on the specification of the intended behavior of the tested program part. Important representatives of black box testing known from imperative programming are

- *specification based testing*: this approach structures the set of valid inputs into partitions, each of which should be considered by test cases.
- *random testing*: a number of appropriate test cases is generated on a random basis. The program specification is just considered to determine the set of valid input values.

Since black box testing does not take into account the tested code, the effect of the used programming language or programming paradigm is not very large. There, however, exists an effect, e.g. regarding the types of the possible test cases. For instance, in functional programming one has to cope with functions as inputs.

4.1.2 White Box Testing

In contrast, *white box testing* does consider the code of the tested program. *Structure oriented testing* (which is the main representative of white box testing) is usually based on a flow graph representing the control and data flow of the tested code fragment. It tries to find a test set that covers all (or at least as many as possible) items of the flow graph. Many different coverage criteria exist, differing in the choice of these items, e.g. all *nodes*, all *edges* or all *definition use pairs*.

When considering structure oriented testing, the usual approaches for imperative programs do not carry over easily to functional ones. For imperative programs the most commonly used coverage criteria are control flow oriented ones.

In functional programming, two main problems arise in applying the usual structure oriented testing methods.

- The flow graph generation is more complicated due to the existence of higher order functions: they cause an influence of the data flow on the control flow in the program. Therefore, data flow analysis is necessary during the flow graph generation.
- The usual control flow oriented coverage criteria do not carry over to functional programming easily because the control structures in modern functional languages differ from the structures known from imperative languages. Furthermore, in lazy functional programs it is complicated to predict the control flow.

At the moment, there exists one approach on structure oriented testing for Erlang programs [Wid05a, Wid05b]. This approach shows the need for performing data flow analysis during the flow graph generation. The data flow analysis aims at

being as powerful as possible in order to provide the best possible accuracy of the flow graphs. There are, however, programming constructs in Erlang that force approximations to take place during the generation of the flow graphs.

4.2 Generation of Test Sets

The generation of test sets can be performed in two ways.

- A test engineer can manually generate the needed tests.
- The tests can be generated automatically by a test case generator.

The manual test set generation is a tedious, time consuming and expensive task. An automatic tool should, hence, be preferred. This problem has already been addressed by the tools QuickCheck [CH00, CH02] and GAST [KATP03, KP05] in the context of functional programming. These approaches use static type information to generate members of a function's input type in a random or systematic manner.

QuickCheck and GAST are test case generators that address the black box testing approach. Both tools are essentially guided by the types of the needed input values. They differ in the employed enumeration strategy for the test cases.

- QuickCheck is a dedicated random testing tool. The predefined generators work on a random basis and the proposals for writing additional generators also address the random testing approach.
- GAST applies a mixed approach for the test case generation: it generates "common border values" and random values of the appropriate types.

The type based, automatic generation of test cases is very helpful in testing functions whose input domain can be described precisely by a type. Both tools have, however, a number of disadvantages.

- Properties that cannot be expressed by a type must be addressed by subsequent filters or by hand written test case generators. While the filters can cause a large fraction of the generated test cases to be dropped, the user provided generators increase the complexity of the testing process for the test engineer.
- Both tools cannot guarantee to generate test cases for specification based testing. For QuickCheck, any structure in the input domain of the tested function is out of the scope of the system. GAST tries to generate border values; it is, however, restricted to borders given by the type information and cannot take into account special border values according to a further program specification.

Up to now, an approach which combines the automatic test set generation and the approaches for judging the quality of test sets is missing.

4.3 Research Directions on the Choice of Test Sets

The question how to choose appropriate test sets for functional programs offers a number of research directions that are worth a further investigation.

In the area of specification based testing some further work is necessary on partitioning the special inputs occurring in functional programming. Input domains containing functions cannot be structured in the same straightforward manner as e.g. numbers. Further investigations should determine, whether the usual form of specification based testing can be used for function type input domains and how these domains can be structured.

Future work in the area of structure oriented testing of functional programs should discover ways of improving the accuracy of the generated flow graphs. Programming languages with static type checking provide assistance by excluding programs with ambiguous data flow and by making further useful information accessible to the data flow analysis process.

Furthermore, excluding some of the programming constructs found e.g. in Erlang programs can greatly improve the precision of the data flow analysis. An identification of problematic programming constructs in different functional programming languages is desirable and can lead to instructions on choosing a functional programming language that is well-suited for structure oriented testing.

In the area of the test case generation, further research should address the automatic incorporation of non-type information into the test case generation process. Candidates for useful information are the following.

- User provided specifications of the input domains of the tested functions should be taken into account. This can e.g. improve the choice of the correct border values.

- The patterns occurring at top level of a function can be used to provide additional systematics to the test case generation. This applies to case distinctions not completely based on type information and to dynamically typed languages in order to replace the non-available static type information. Such an approach can be a first step of considering structure oriented information for the automatic test case generation.

- The combination of automatic test case generation with structure oriented judgement of the test cases can improve the test case generation. Further research is necessary to see how information from the structural testing can be fed into a test case generator and how they can be used to compute test cases appropriate for not yet covered program parts.
  According to existing publications on generating test cases according to the edge coverage criterion in imperative languages [ABB+91],[BKM91], the fully automatic coverage of a program seems to be infeasible. A successful development in this area must find a trade-off between the fully automatic coverage of the tested code and the feasibility of the test case generation process.

Although tools with these extensions will no longer be lightweight, as claimed for the existing approaches on automatic test generation, we expect these directions of research to lead to more powerful yet easily usable testing tools.

5 CHECKING THE RESULTS FOR CORRECTNESS

When a test case has been evaluated the evaluation result needs to be checked for correctness. The following main approaches are possible to do this.

- A test engineer manually checks the output and compares it with the output he/she expects. The test case, the computed result and the judgement of the result are stored in a data base for future use.
- The test system judges the correctness of the result automatically, based on a number of predicates that must be fulfilled for the result to be correct.
- In regression testing a piece of code is tested again after changes have been made. The previously used test cases are reused, and the correctness check is done by comparing the new results with the previous ones.

Regression testing is a special testing method whose processing depends on the previously applied testing approach. When automatic predicate analysis has been used before, this approach can simply be repeated. In case of manual judgement of the original test results, the succeeding test cases of the former testing can be performed automatically by comparing the current results with the previous ones. The tester just needs to intervene for the previously failed test cases and for the cases with discrepancies between the current and the previous result. Since regression testing does not need a special treatment, it is not considered any further in the rest of this section.

In all testing situations, a test that is not considered correct gives evidence for an error, that needs to be discovered, in the tested program.

5.1 Manual versus Automatic Checking

The testing approaches mentioned above have the following advantages and disadvantages.

For the manual check the tester directly transforms the informal specification of the intended program behavior into a mental model, which is used in generating an expectation on the program result. Either he can check the outcome of the test directly with his expectation, or he stores all expected outcomes together with the corresponding test cases and lets the system perform an equality check automatically. The main advantage of the manual check is that the computed result is directly compared to an intuitive expectation without a formalization (which could contain errors itself) in-between. The main disadvantage of this approach is its time consumption and cost.

Checking results automatically against formalized predicates has the main advantage of a reduced time consumption and cost. As a disadvantage, the formal specification given by the predicates can be erroneous or incomplete. Especially in the case of incompleteness this problem does not become obvious easily.

In both cases, the testing (checking of the results or providing the necessary predicates) should be performed by a person different from the programmer. In this way, misinterpretations of the program specification which caused errors in the program are less likely to be repeated during the testing.

5.2 Equality of Expected and Computed Results

The approach of providing an expected result with the test input and to check automatically whether the expected and the computed results are equal is a very common method of checking the correctness of test results. In fact, the use of this method is common enough to find a special construct for equality checks in testing frameworks like HUnit [Her02]. The equality check also occurs regularly when applying regression testing.

In functional programming the situation is complicated by the fact that the result values can have types not allowing an equality check. Among others this affects results in the form of

- functions
- infinite lazy structures

For these result types the checking of the correctness is generally problematic. For both types of results, it is not only impossible to check them for equality, but a full inspection (either automatically by predicates or manually by a human tester) is also impossible.

Results of these types can be checked only partially. The checking has to be performed to an extend that gives enough evidence for the correctness of the result.

Further problems in checking equality occur, when there are several possible results that are equivalent. E.g. when lists are interpreted as sets, there are several lists in general that contain the same values and that should all be interpreted as correct.

User provided equality predicates that are specialized to the expected output types can be of help here. Some ideas on this topic have already been presented for testing students' programs in the teaching context with a specimen program available [Wid04b].

5.3 Research Directions on Correctness Checking

In checking the testing results for correctness there are two possible directions of further research.

- For the tools providing automatic checking based on predicates: how is the typical quality of the testing predicates and how can it be improved?
- Can we find some standard methods to test complex outputs like functions and infinite structures?

For the moment it is up to the test engineer to provide predicates which test all of the expected properties of the output values. From the author's experience, however, some of the properties seem to be less natural than others and are overlooked more easily. Case studies of testing larger software projects based on predicates should investigate whether there are certain properties of output values that are forgotten frequently and how the process of providing appropriate predicates can be supported.

The inspection of functions and infinite structures returned as output values must naturally remain incomplete. For infinite structures the situation is quite simple as inspecting and comparing a sufficiently large prefix of the structure usually gives enough evidence for the correctness of the structure. Automatic tool support for this common situation should be easy to provide.

The situation in checking functions is more complicated. A result function f can only be inspected by providing appropriate inputs. To complicate the problem even more, the types (and further properties) of the new inputs for f depend on the test case that generated f. In general, a new test set can be provided for each of these f. Since f can, however, be a higher order function itself the test sets can become complex and confusing quite soon. New approaches for a succinct and understandable management of the test sets are necessary and need further investigation.

6 CONCLUSION

Testing is an important part of each software development process, no matter which programming paradigm is used. In functional programming its low level nature caused missing acceptance of the software testing by parts of the community. Publications from the last years show that testing of functional programs has (eventually) received some more attention.

In this work we have described the differences between several approaches to software quality based on their goals and their applicability. We showed that none of the other approaches can be used as a full replacement for testing. This motivated the statement that the approach of software testing should be considered carefully in the functional programming paradigm.

We have identified the testing of small program fractions as mostly dependent on the used programming paradigm. Here, the choice of appropriate test data and the question whether a generated result is correct are the areas that are influenced by the use of functional programming languages.

For both of these areas, we have identified the main issues related to the testing of functional programs. Based on this identification, an overview over existing

work has been presented and we have discussed open problems and possible directions of future research.

The choice of test cases by structure oriented testing could benefit from improvements of the flow graph generation by optimal precision in the data flow analysis. For the automatic generation of test cases we suggest research on the incorporation of non-type information such as precise specifications of the intended program properties, the top level patterns of the tested functions or information from a structural coverage test.

For the judgement of test results, two directions of future research have been identified: information on the implementation of high quality predicates for the automatic correctness checking of complex functions is missing. Case studies should investigate common problems and give standard recommendations for implementing such predicates. For all types of correctness checking (manual and automatic) further research could provide standard methods for the correctness analysis of complex data structures and especially of functions.

Altogether, testing of functional programs is an area which is worth consideration and which contains several open questions for a further investigation.

REFERENCES

[ABB+91] A. Auzins, J. Bārzdins, J. Bicevskis, K. Cerans, and A. Kalnins. Automatic construction of test sets: Theoretical approach. In J. Bardzdins and D. Bjorner, editors, *Baltic Computer Science*, pages 286–359. Springer, April 1991.

[BKM91] J. Borzovs, A. Kalnins, and I. Medvedis. Automatic construction of test sets: Practical approach. In J. Bardzdins and D. Bjorner, editors, *Baltic Computer Science*, pages 360–432. Springer, April 1991.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., September 18–21 2000. ACM Press.

[CH02] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 65–77, New York, NY, USA, 2002. ACM Press.

[Chi01] Olaf Chitil. A semantics for tracing. In *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL*, 2001.

[Gil00] Andy Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham*, 2000.

[Her02] Dean Herington. *HUnit 1.0 Users Guide*, 2002. http://hunit.sourceforge.net.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[KATP03] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. GAST: Generic automated software testing. In *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100, 2003.

[KP05] Pieter Koopman and Rinus Plasmeijer. Generic generation of elements of types. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[Lig02] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[Nai97] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[Rak97] Steven R. Rakitin. *Software Verification and Validation*. Artech House, Inc, 1997.

[WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy*, pages 151–170, 2001.

[Wid04a] Manfred Widera. Flow graphs for testing sequential Erlang programs. In *Proceedings of the 3rd ACM SIGPLAN Erlang Workshop*. ACM Press, 2004.

[Wid04b] Manfred Widera. Testing Scheme programming assignments automatically. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 4. Intellect, 2004.

[Wid05a] Manfred Widera. Concurrent Erlang flow graphs. In *Proceedings of the Erlang/OTP User Conference 2005*, Stockholm, 2005.

[Wid05b] Manfred Widera. Data flow coverage for testing Erlang programs. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[XP06] *Extreme Programming: A gentle introduction*, 2006. www.extremeprogramming.org.