

Towards Flow Graph Directed Testing of Functional Programs

Manfred Widera

Fachbereich Informatik, FernUniversität Hagen
D-58084 Hagen, Germany
Manfred.Widera@fernuni-hagen.de

Abstract. Testing of software components during development is a heavily used approach to detect programming errors and to evaluate the quality of software. Systematic approaches to software testing get a more and more increasing impact on software development processes.

In the context of imperative programming there are several different approaches to measure the appropriateness of a set of test cases for a program part under testing. Some of them are source code directed and given as coverage criteria on flow graphs. Source code directed means that a test set is considered to be more appropriate if it covers more statements, paths, . . . in the source code or flow graph of the tested module.

The aim of this paper is to provide a definition of flow graphs for the sequential part of Erlang hereby providing a first step towards the transfer of the usual source code directed testing methods to functional programming.

1 Introduction

Testing of software is a widely used method of detecting errors during the software development process. One can assume every software to be tested before being put to use in practice. Though testing can just prove the presence but not the absence of errors, the passing of all tests given by an appropriate test set is often understood as an evidence for reaching a certain level of software quality. For imperative programming there are several approaches defining the appropriateness of a test set by coverage criteria based on the flow graph. Testing in this way is usually applied to small program fractions like single modules.

In the context of functional programming there is up to now no method for measuring the quality of a test set that takes the program structure into account. As systematic testing becomes a more and more important task of industrial software development, it is crucial to present source code oriented testing methods for functional programming languages in order to increase their acceptance in industry. Especially, systematic testing cannot be replaced by employing the suitability of functional languages for verification, because of two reasons. First, verification is a quite time consuming task and cannot be applied to all the less

critical components (which should nevertheless be as correct as possible). Second, verification is always done against an already formalized specification of the intended program behavior which itself need not be correct.

The aim of this paper is to give a definition of control flow graphs of functional programs (more precisely of programs written in the functional language Erlang [1] which was chosen because of its already given relevance in industry) similar to the known flow graph definition for imperative programs. This forms the first step towards making the large area of systematic, source code directed testing available for functional programs. The use of the flow graph definition is shown by carrying over the definition of some simple coverage criteria without much effort. Further work is, however, necessary to verify the use of the different known coverage criteria for functional programs as new application.

The rest of this paper is organized as follows: in Sec. 2 we discuss some work related to our approach. The restricted language we work on and the definition of flow graphs for this language are given in Sec. 3. Section 4 provides some information on data flow analysis in functional flow graphs which is needed in Sec. 5 for generating flow graphs for higher order programs. In Sec. 6 a short overview over the opportunities and problems of flow graph coverage by test cases is given. In order to transfer this work to other functional languages, some problems have to be addressed which are briefly discussed in Sec. 7. Section 8 completes the paper with a conclusion and some notes on future work.

2 Related Work

The work presented here is related to publications from several areas. There are already approaches on flow graphs for functional languages. In [11] flow graphs and call graphs are used in the context of software measurement for functional programs. The flow graphs used there consider function calls as atomic operations and are generated for each function independently. We will show later that our approach can be understood as an integration of flow graphs for single functions and a call graph into a single structure which is able to express the control flow in recursive function definitions.

The concepts of generating flow graphs for higher order programs can be found e.g. in [10], [2]. Indeed our approach is equivalent to OCFA, but has a focus on the presentation of the results to human programmers. In detail, we avoid program transformations using continuation passing style (CPS) and the Y combinator but work on a version of the program that only slightly differs from the source code it was derived from.

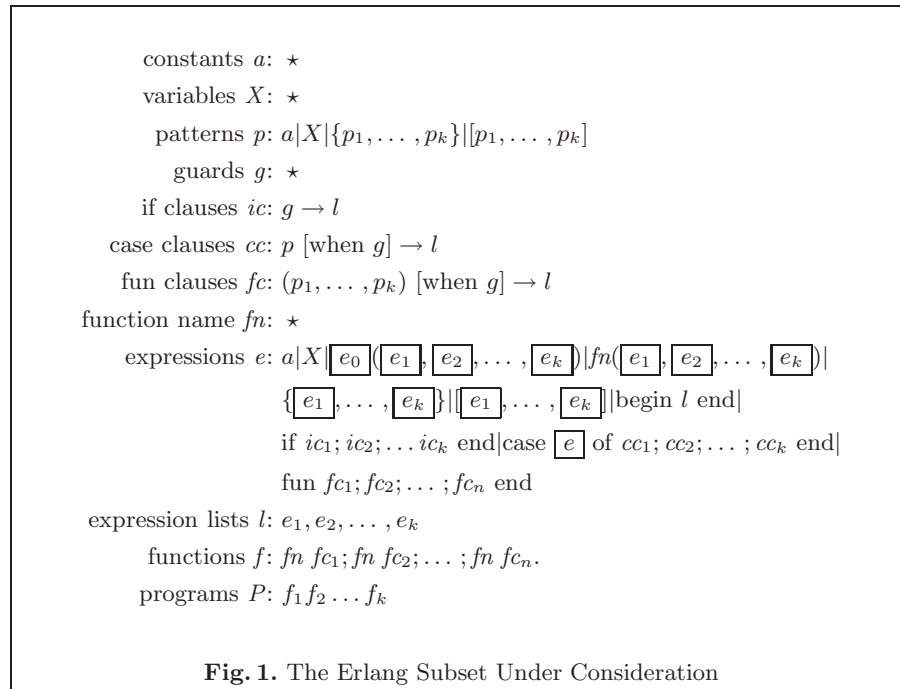
Furthermore, different approaches on testing and debugging functional programs have been proposed. QuickCheck [4] aims at automatically checking Haskell programs by generating input data on a random basis and checking the results with constraints on the expected output. In [8, 9] flow graphs are used for judging the coverage of a functional program by a set of test inputs. This approach is, however, restricted to spread sheets considered as first order functional programs without recursion. Several approaches on declarative debugging and trac-

ing functional languages (e.g. [7], [5], [3,12]) describe how to trace down the programming errors causing an observed misbehavior of a program. These approaches, however, do not provide mechanisms for generating or judging the test sets that are used to provoke such a misbehavior.

3 Preliminaries

3.1 Restricting the Language Under Consideration

For presenting the definition and generation of functional flow graphs we do not consider the whole Erlang language. We rather concentrate on the subset defined in Fig. 1 (ignoring the boxes around some expressions for the moment). Definitions consisting of a \star are not needed in this work and therefore omitted here. Infix operators are considered as ordinary functions in this definition. In the following when speaking of a first order function call we mean a call of the form $fn(e_1, e_2, \dots, e_k)$ with a function name fn , and a higher order function call has the form $e_0(e_1, e_2, \dots, e_k)$.



Programs using this Erlang subset are normalized to meet the following properties:

1. Every function just consists of *one* clause with only variables as arguments. (This guarantees a unique entry point for every function.)
2. All function calls, operator applications, data constructors, data selectors, and conditionals essentially just have variables as arguments. (This provides names for all argument values and makes the evaluation of the arguments *before* the call explicit.)

Condition (1) is met by introducing a new form of expressions generated by the keyword *funcase* of the form

funcase ($\boxed{e_1}, \dots, \boxed{e_k}$) of $fc_1; fc_2; \dots; fc_k$ end

and replacing a function definition $fn\ fc_1; fn\ fc_2; \dots; fn\ fc_n.$ by

$fn\ (X_1, \dots, X_k) \rightarrow$ funcase (X_1, \dots, X_k) of $fc_1; fc_2; \dots; fc_n$ end.

where X_1, \dots, X_k denote fresh Erlang variables.

To meet Condition (2) we replace all expressions that are boxed in Fig. 1 by variables and introduce the necessary variable bindings before the containing expression. For e.g. list constructions this means replacing $[e_1, \dots, e_k]$ by

$X_1 = e_1, \dots, X_k = e_k, [X_1, \dots, X_k]$

where X_1, \dots, X_k are fresh variables. The other expressions containing boxed subexpressions (including the e_i in the list example) are processed analogously.

We will illustrate the program transformation with an example that will be revisited again for describing the further steps of generating a first order functional flow graph.

Example 1. Consider the following definition of the functions even and odd.¹

```

even(0) -> true;           odd(0) -> false;
even(N) -> odd(N - 1).    odd(N) -> even(N - 1).

```

Performing the described preprocessing steps results in the following function definitions:

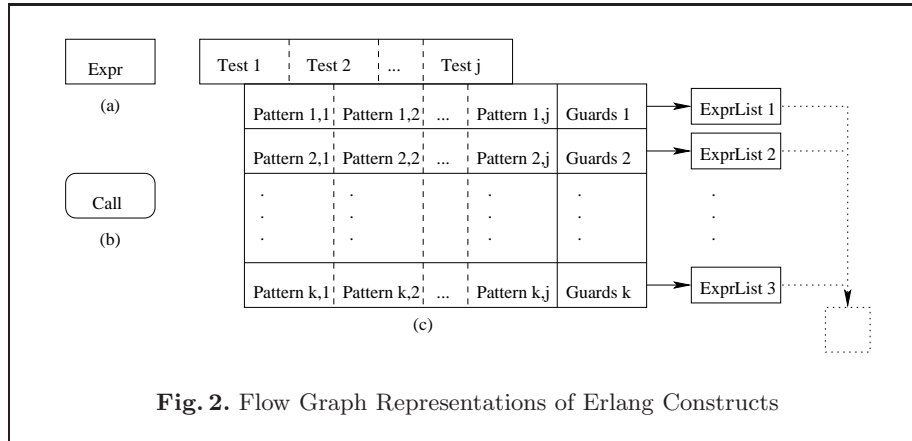
```

even(Arg1) ->                odd(Arg2) ->
  funcase Arg1 of            funcase Arg2 of
    0 -> true;                0 -> false;
    N -> Var1 = N - 1,        N -> Var2 = N - 1,
        odd(Var1)              even(Var2)
  end.                        end.

```

3.2 Flow Graphs for Erlang

Flow graphs for programs using the language of Subsec. 3.1 consist of representations of the individual expressions as nodes and edges representing the control flow between these nodes.



There are different forms of nodes in a flow graph representing different kinds of expressions. These node forms are presented in Fig. 2.

- For computations by operators, data constructors, data selectors, BIFs, and other functions not represented in the flow graph, a node is generated as shown in Fig. 2.a.
- Function calls to a function represented in the flow graph are expressed by nodes as in Fig. 2.b.
- *case* expressions are represented as in Fig. 2.c with $j = 1$. The test expression to be matched against the patterns is stored in the **Test** field. The **Pattern** fields hold the patterns of the individual clauses, the **Guards** fields the list of guards and the **ExprList** fields the graphs of the list of expressions evaluated when the corresponding clause is chosen.
- *if* expressions are represented as in Fig. 2.c with $j = 0$. The box containing the **Test** remains empty, the boxes for the **Pattern** fields are omitted.
- *funccase* expressions are represented as in Fig. 2.c where j equals the arity of the function.

The set of nodes representing a function definition is extended by:

1. an import node for f defining of the argument variables of f .
Notion: $\text{import}(a_1, \dots, a_k)$, where the a_i are the argument variables imported by the function.
2. a context node for f providing a local definition for all variables that are not defined within the function, but come from the context f was defined within. (As long as we just consider first order functions, the context node is always empty, i.e. does not contain any variable definitions. Context nodes

¹ In the examples throughout this paper, we omit module declarations and imports/exports of functions, as long as they are not necessary for the understanding.

are needed in the definition of funs.²)

Notion: $\text{context}(v_1, \dots, v_k)$, where the v_i are the variables whose definitions are taken from the context of the function definition and which are used within the function.

3. a return node for f expresses leaving the code of f and returning the calculated value given in the variable R .

Notion: $\text{return}(R)$.

Within a single function directed edges are introduced from node n_1 representing an expression e_1 to node n_2 representing a node n_2 if one of the following conditions hold:

- n_2 follows n_1 in the function's source code
- n_1 is of the form (c) in Fig. 2 and e_2 is the first expression in a body of a corresponding clause (note that in this case n_1 has a complex form and the source of the edge must be connected to the right pattern/guard)
- e_1 is the last expression in the body of a clause and e_2 is the expression following the conditional containing this clause.

For flow graphs of individual functions an example is given in Ex. 2.

Example 2. The flow graphs for the two functions *even* and *odd* defined in Ex. 1 (after preprocessing) are given in Fig. 3.

For function calls we have the following control flow: when reaching a node n representing a call to a function f then the control is transferred to the import node of f . When reaching the return node of f , the control jumps back to the (unique) node n' following n in the single function flow graph described above.

A naive approach describing this control flow by edges is the following:

- An edge from n to the import node of f is introduced.
- An edge from the return node of f to n' is introduced.
- The edge from n to n' is deleted since there is no direct control flow between n and n' .

Unfortunately, this approach has the following disadvantages:

- The large number of new edges, especially the return edges make the resulting flow graph quite complex.
- By deleting edges in the flow graphs of the individual functions, program expressions that are related in the source code and in the program semantics are not connected via a short path in the flow graph any more.

² In Erlang funs are special function representations that allow functions to behave as values. They are distinguished from ordinary function names because of historical reasons. In principle, funs in Erlang behave as λ -expressions or λ -closures in other functional languages.

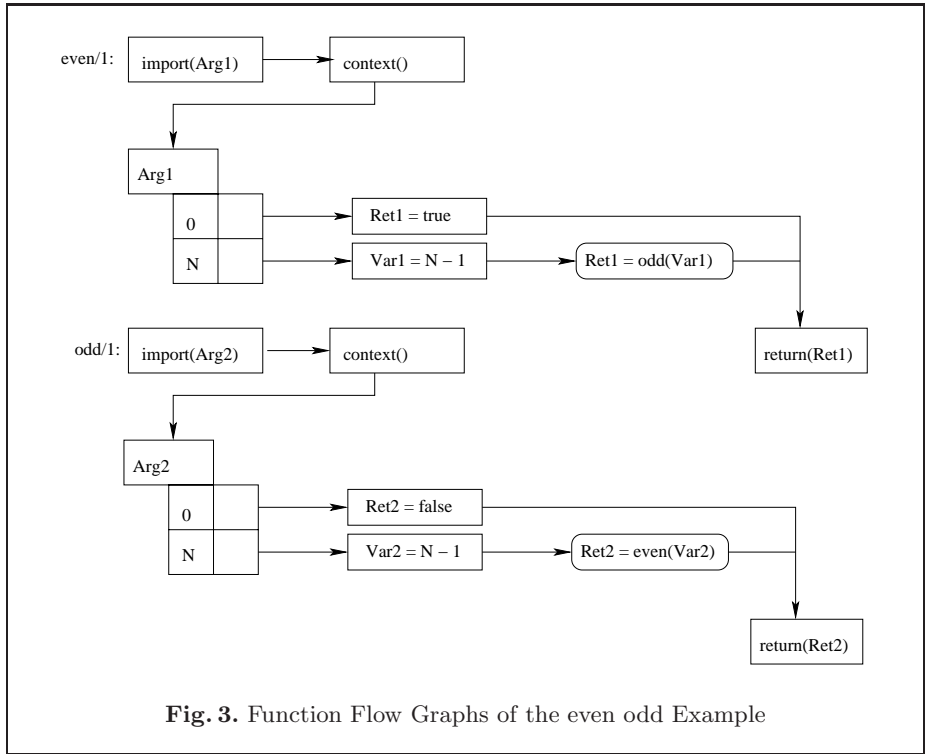


Fig. 3. Function Flow Graphs of the even odd Example

To prevent these disadvantages, we introduce a new notion of *call edges* or *rubber band edges* into functional flow graphs. Such a rubber band edge (going from the calling node to the import node of the called function in the flow graph) represents the control flow to the called function and the return after reaching the return node of this function. (We can understand a function call in the flow graph as tying the arguments of a call to one end of a rubber band and throwing them to the called function while holding the other end of the band. When the computation of the call finishes, the band bounces back with the result tied to it and the computation goes on with the next node in the local flow graph.)

Using rubber band edges we can revisit the problem of processing calls in node n to a function f . Such a call is represented by a rubber band edge from the node n to the import node of f .

In order to argue on the individual elements of a functional flow graph G we define the notion $G = (V, E, C)$, where V denotes the set of all nodes, E the set of all edges within a single function and C the set of all call edges.

Example 3. For the even odd example given in Ex. 1 we get the flow graph presented in Fig. 4. (Rubber band edges are denoted by dotted lines.)

Comparing our functional flow graphs with [11], we defined an integrated structure formed from the flow graphs for individual functions (extractable by

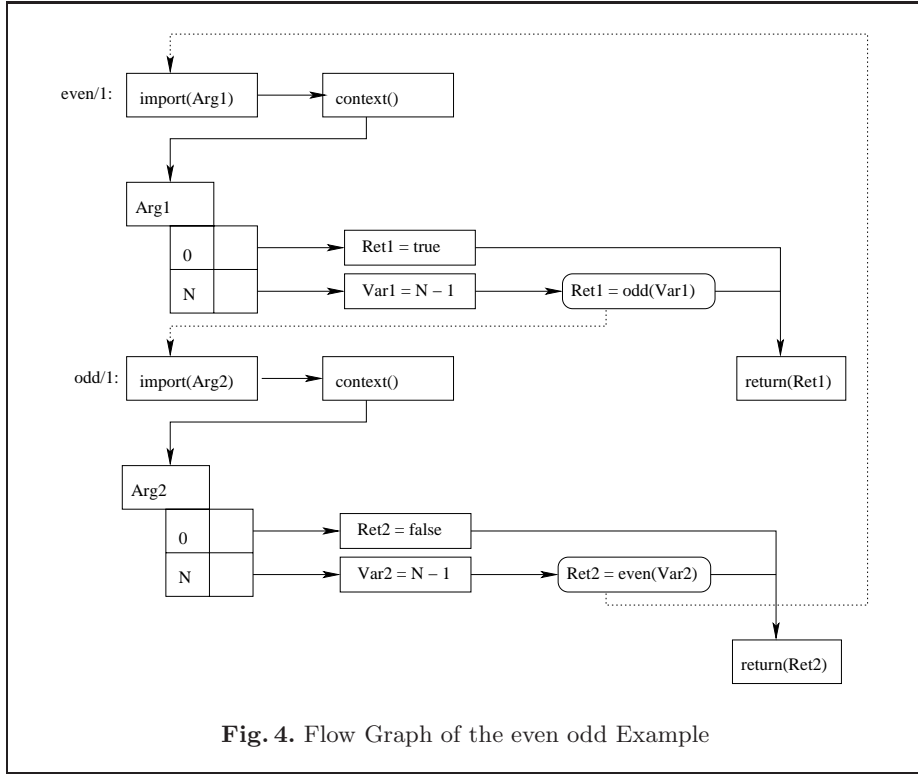


Fig. 4. Flow Graph of the even odd Example

deleting the call edges) and the call graph of the program (given by the call edges when compressing every function flow graph to a single node).

4 Data Flow Analysis in Functional Flow Graphs

As known from [10], the control flow given by a higher order program can depend on the data flow of higher order function definitions. In this section we therefore give a definition of some base notions for data flow analysis.

First of all, functional flow graphs contain definitions and uses similar to those given in imperative languages. For a definition of a variable v we write $def(v)$, for a use of v we write $use(v)$. Their precise definition is as follows:

Definition 1 (Definitions). *Let G be a flow graph and v a variable. A node n in G contains a definition of v if one of the following conditions holds:*

- n is an import node and v is one of the variables defined in n .
- n is a context node and v is one of the variables defined in n . This is called an f -definition and denoted by $f-def(v)$.

- n denotes a pattern matching $LHS = RHS$, v occurs in LHS , and there is at least one path p from the beginning of the function containing n to n itself such that v is not defined on w .
- n denotes a conditional, v occurs in at least one pattern p in n , and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on p . A specification of such a definition, besides the node n , contains the number of the clause, the pattern p belongs to. Occurrences of v in the patterns of several clauses of n are treated independently.

A definition binding v to a value selected from a list or tuple (either by pattern matching or the corresponding selection BIFs) is called an *s-definition* and denoted by $s-def(v)$.

Definition 2 (Uses). Let G be a flow graph and v a variable. A node n in G contains a use of v if one of the following conditions holds:

- n is a node representing the expression E or a match $p = E$ where
 - $E = v$
 - $E = \{v_1, \dots, v_k\}$ or $E = [v_1, \dots, v_k]$ with $v = v_i$ for some i . This is called an *s-use* and denoted by $s-use(v)$.
 - $E = v_0(v_1, \dots, v_k)$ or $E = fn(v_1, \dots, v_k)$ with $v = v_i$ for some i .
- n is a conditional node where one of the tests is given by v .
- n contains the generation of a fun, and there is at least one path p from the beginning of the function containing n to n itself such that v is defined on p . This is called an *f-use* and denoted by $f-use(v)$.
- n denotes a pattern matching $LHS = RHS$, v occurs in LHS , and there is at least one path p from the beginning of the function containing n to n itself such that v is defined on p .
- n denotes a conditional, v occurs in at least one pattern p in n , and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w . A specification of such a use, besides the node n , contains the number of the clause, the pattern p belongs to. Occurrences of v in the patterns of several clauses of n are treated independently.

The special definitions and uses defined above have the following meaning:

- An *f-use* uses a variable to freeze it in a function definition until it is defrosted by an *f-definition* in the corresponding context node.
- An *s-use* uses a variable to store its value in a structure, where it can be taken from an a following *s-definition*.

For both, *f-definitions* and *f-uses* as well as *s-definitions* and *s-uses*, we need to define the notion of *corresponding* uses and definitions.

Definition 3 (corresponding *f-use*, *f-def*). Let v be a variable, u an *f-use* of v , and d an *f-definition* of v . u and d correspond to each other if the fun containing d is the one defined in u .

Definition 4 (corresponding s-use, s-def). *Let v be a variable, u an s-use of v generating a structure c . A selection d defining a variable v' is an s-definition of v' corresponding to u if the structure decomposed in d is c , and the selected element position is the one containing the value of v .*

Note that for an s-use and the corresponding s-def the variable names can differ.

The next definition states the situations under which a definition d reaches a use u . This definition especially takes into account freezing, the packing into a structure, and the renaming of values by copying them into new variables.

Definition 5. *Let d be a definition of a variable v , and u a use of a variable v' . Then d reaches u if one of the following properties holds:*

- $v = v'$ and there is a path in the flow graph from d to u that does not contain a definition of v different from d . In this case we say d reaches u directly.
- There is a copy expression e of the form $\tilde{v} = \tilde{v}'$ such that d reaches the use of \tilde{v}' in e and the definition of \tilde{v} in e reaches u .
- d reaches an f-use of v and there is a corresponding f-definition of v that reaches u .
- d reaches an s-use of v and there is a corresponding s-definition of some v' that reaches u .

Note that each rubber band edge from a function call c to a function f/k contains implicit assignments $p_i = a_i$ for each $i = 1, \dots, k$ where p_i is the i^{th} parameter in the definition of f and a_i is the i^{th} argument in the call c . These implicit assignments are processed like ordinary renamings according to Def. 5.

5 Flow Graph Extension for Higher Order Programs

For higher order programs the generation of a flow graph becomes more complicated than in the first order case because of the following fact already mentioned in [10]: since functions can be passed around as arbitrary values, the control flow possibilities at a certain program point can depend on the flow of data to this point. This is especially the case for calls where the called function is given by a variable. In this case the further possible control flow options are given by the definitions that reach the use of this variable in the call.

For generating this data flow information and therefore for generating the flow graph of a higher order program we already need a flow graph. Fortunately, we can solve this problem by an iterative process starting with the first order flow graph and introducing higher order call edges iteratedly. The details of this process are defined as follows:

Definition 6 (first order flow graph). *Let P be a program. The first order flow graph $G_1(P)$ of P consists of:*

- *The flow graphs of all functions defined in P (including those generated by fun-expressions.)*

- The call edges for all first order calls in P .

The generation of $G_1(P)$ for a given program P is straightforward.

Definition 7. Let P be a program and $G = (V, E, C)$ a flow graph of P with potentially missing call edges for higher order calls. We define an operator Δ by

$$\Delta(G) = (V, E, C \cup C')$$

where C' contains all call edges from a node n to the import node of a function f with:

- n denotes a higher order call with the called function given by a variable v .
- Let u denote the use of v in n . There is a definition d of a variable v' such that d assigns the function f to v' and d reaches u in G .

For a higher order Erlang program P we now have the following constructive definition of the flow graph of P :

Definition 8. Let P be program, $G' = G_1(P)$ the first order flow graph of P . The flow graph G of P is the least fixed point of Δ when applied to the initial value G' .

In principle, the generation of the flow graph G of a given program P is equivalent to OCFA [10]. In contrast to the approach presented there, our work is adapted towards presenting the results to human users. This includes to pre-process the program source code as little as possible. Furthermore, we use a refined notion of definitions reaching uses that helps to avoid escapes of used functions through structures.

Example 4. Consider an Erlang module consisting of the following definitions:

```
list_add_1(L) -> map(fun add_1/1, L).

add_1(N) -> N + 1.

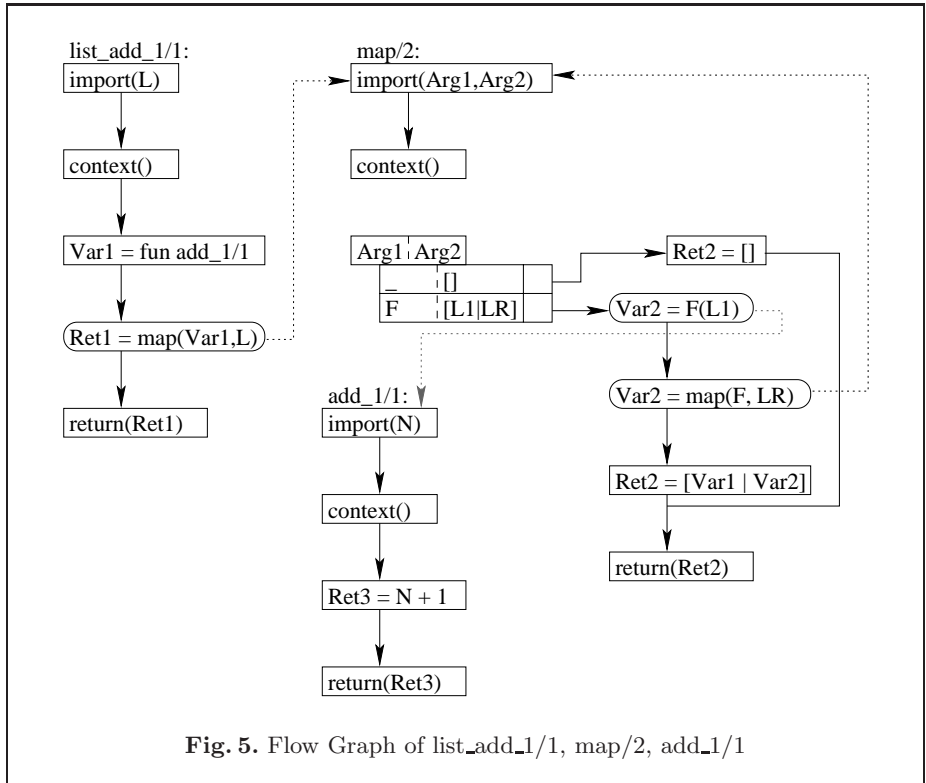
map(_, []) -> [];
map(F, [L1|LR]) -> [F(L1) | map(F, LR)].
```

Preprocessing yields the following function definitions:

```
list_add_1(L) -> Var1 = fun add_1/1,
                 map(Var1, L).

add_1(N) -> N + 1.

map(Arg1, Arg2) ->
  funcase Arg1, Arg2 of
    _, [] -> [];
    F, [L1|LR] -> Var2 = F(L1),
                  Var3 = map(F, LR),
                  [Var2 | Var3].
```



The flow graph of this module is given in Fig. 5. Note that the rubber band edges drawn in black are introduced while generating the first order flow graph, but the edge drawn in grey is introduced by data flow analysis.

6 Test Case Analysis Using Flow Graphs

For the flow graph of a sequential Erlang program as defined above, we now want to discuss its use for evaluating test cases. Test case evaluation is usually done using a tool that records all the visited nodes (and edges) during executing tests. Subsection 6.1 contains the discussion of some special properties of the control flow in functional languages and discusses how these specialties influence the execution of test cases and the recording of visited nodes.

In Subsec. 6.2 we will briefly discuss how some of the simple coverage criteria of flow graphs known from imperative programming can be transferred to functional flow graphs.

6.1 Specialties of Functional Control Flow

No matter which of the coverage criteria described above is chosen, the execution of the tests has to be performed in a supervised manner in order to collect the information about the followed execution paths.

A naive approach that works fine for imperative and first order functional programs is an interpreter that executes the tests on the flow graph hereby simulating the program execution in the usual runtime environment. Each call to an unsupervised program part (either to a module that was tested independently or a library function) is passed to the runtime system for evaluation.

For higher order programs this approach is no longer suited because of the following problem: when a function f under supervision is passed to a library function g as argument then the evaluation of the call to g including calls to the argument f is performed by the runtime system. For f it is therefore possible to be executed without supervision.

Example 5. Consider a module just consisting of the following two function definitions.

```
list_add_1(L) -> map(fun add_1/1, L).
```

```
add_1(N) -> N + 1.
```

When we further assume that this module is the only one under supervision (especially that the library module lists containing `map/2` is not) then no test on `list_add_1/1` will result in a coverage of `add_1/1`, although it is covered when calling `list_add_1/1` with a non empty list of numbers as argument.

Compared with the flow graph of Ex. 4 the function `map/2` and all edges connected to it are missing in the flow graph of the module discussed here. There is, hence, no connection between `list_add_1/1` and `add_1/1`.

The following solutions of this problem are possible for higher order functions:

- One can accept the weakness of the interpretation approach discussed above. It can cause program parts to be marked as uncovered although they are covered by function calls from outside the supervision. If the system states a coverage criterion fulfilled, this information is, however, reliable.
- An interpreter as described above can be used not only for the program parts under supervision, but also for the rest of the program. This gives full control over the execution and supervision of the program to the interpreter. For this approach it is crucial that all library modules used in the program (even modules that are just used by unsupervised modules) are available in source code.
- The runtime system can be modified to pass the control back to the interpreter when calls to supervised functions occur from outside the supervision. This approach is the most promising one since it causes the fewest restrictions for the user of the system.

6.2 Transfer of Known Coverage Criteria

In the research of testing imperative languages there exist several different coverage criteria on flow graphs. Each of these criteria defines a set of entities in the flow graph that should be covered in the best possible way by the set of performed tests.

Following [13] we want to discuss some of the criteria and their use for functional programming.

- The node coverage criterion requires all *nodes* in the flow graph to be covered at least once. This corresponds to the execution of every statement in imperative programming, and to the evaluation of every expression in functional programming.
- In the branch coverage criterion every *edge* in the flow graph must be covered. When considering first order functional programs, this is equivalent to the node coverage criterion. For higher order programs this criterion especially enforces every supervised higher order function to be called at least once in every possible supervised position.
- Criteria based on the coverage of conditions seem not appropriate for functional programming in Erlang. This is the case, because in the context of Erlang programs, the form of conditions is determined by pattern matching.

In [6] Liggesmeyer states, that for object oriented programming, we can use the same testing tools as for structured programming, but the appropriateness of the individual tools changes and therefore should be revalidated in the new framework. We can assume that the same holds for the functional programming approach, and that especially the methods based on the control flow of a program are still useful, because functional programs (in contrast to many object oriented programs) have a clear control flow. Further work is possible and necessary in this area to investigate the use of criteria that are based on covering larger paths in a program (an open question here is the influence of recursion on these criteria) and criteria based on the data flow.

7 Transfer to Other Functional Languages

The main concepts described in this paper so far are relevant for other functional programming languages, as well, although some of the decisions in detail are focussed on Erlang only. In order to adapt the generation of functional flow graphs to other languages the following topics have to be addressed:

- Pattern matching and branching constructs slightly differ between the different languages. Appropriate flow graph representations for the language under consideration have to be defined.
- Most functional programming languages provide a construct for local variable definitions (usually defined by the keyword *let*). This allows for shadowing variables, i.e. the value bound to a variable before the let expression is hidden during the let expression but available again afterwards. Data flow analysis has to be extended to cope with this shadowing.

- In flow graph generation and flow graph covering, lazy evaluation is a topic. One solution to this problem is to generate the flow graph as before. Variable assignments now do not mark the actual computation, but the generation of a hook for performing the computation afterwards. A node is marked as covered only if it is reached by the control and the defined value is again used in some computation (that must be marked as used again). For nodes that are reached by the control but whose values are not used, a new state between covered and uncovered can be introduced.³

8 Conclusions and Future Work

By adapting the notion of flow graphs to functional programs written in a sequential subset of Erlang we have made a large step towards having the wide area of source code directed testing (which is heavily used in industry) accessible for functional programming.

Function calls have a strong influence on the control flow in functional programs comparable to the looping constructs in imperative languages. We, therefore, had to find a notion of expressing the control flow of a function call, namely the jump to a distant piece of code and the return to the calling code piece after processing the function call. The introduction of rubber band edges allows to express this situation without making the generated flow graphs unnecessarily complex.

When considering higher order functional programs, we get the additional problem that we need data flow analysis in order to determine the set of functions that is possibly called at a certain program point. We have introduced definitions and uses of a value and a notion of a definition reaching a use, that is precise enough to compute all functions that can reach a function call. Furthermore, an iterative process, essentially equivalent to OCFA [10], allows to use the already generated flow graph for the data flow analysis task and to update the graph correspondingly.

The concepts described here are not restricted to Erlang, but should be applicable for other functional programming languages as well. An overview over the topics that must be addressed when transferring the notion of functional flow graphs to other languages has been given.

Future work will complete a testing tool for Erlang programs comparable to those tools already used for imperative programming. Further steps in developing such a system are the extension of the flow graph definition to full Erlang including several processes and process communication, other structures used in Erlang, and the exception handling system of Erlang. For the completed flow graph definition we need to implement a tracing tool for the test execution that copes with the problem of higher order functions escaping the control.

³ This approach may also be of interest for eager functional languages if we are not only interested in expressions being evaluated (and the result potentially thrown away) but in expressions which are evaluated and which results are indeed used to calculate the result of the computation.

With these areas completed, the adaption of different coverage criteria will become an interesting area of research: simple node or branch coverage can already be of great help in keeping the overview over testing complex and nested case distinctions. We can, however, expect further criteria that are well adapted to functional programming and that greatly increase the power of source code directed testing of functional programs.

References

1. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996.
2. J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.
3. Olaf Chitil. A semantics for tracing. In *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL*, 2001.
4. 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.
5. Andy Gill. Debugging haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham*, 2000.
6. Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.
7. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
8. Gregg Rothermel, Lixin Li, Christopher DuPuis, and Margaret Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 198–207. IEEE Computer Society Press / ACM Press, 1998.
9. Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239. ACM Press, June 2000.
10. Olin Shivers. Control-flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.
11. Klaas van den Berg. *Software Measurement and Functional Programming*. 1995.
12. 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*, 2001.
13. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.