

Master Thesis

Using QuickCheck and Semantic Analysis to Verify Correctness of Erlang Refactoring Transformations

Elroy Jumpertz

elroy.jumpertz@student.ru.nl

Department of Computer Science
Radboud University Nijmegen

Supervisor: Rinus Plasmeijer

Document number (RU): 629
Project ID (ELTE): KMOP-1.1.2-08/1-2008-0002

Abstract

A refactoring transformation performs changes on source code that modifies its structure, but keeps its semantics intact. When developing refactoring algorithms one must make sure that the transformation does not change the semantics of the source code. Additionally, the source code must not be changed in such a way that the refactored program becomes syntactically invalid. In this thesis, the correctness of various refactoring transformations (carried out on Erlang code) is examined by using QuickCheck in combination with a newly developed semantic evaluator.

The proposed solution traverses the semantic graph, created by the Erlang refactoring tool RefactorErl, and builds semantic representations of modules. Two functions are considered to be semantically equivalent if the results after evaluation are identical. QuickCheck is used to automatically apply an arbitrary number of random refactoring transformations on a static Erlang module. The semantic evaluator is used to verify the meaning-preserving behavior of the refactoring transformations under test.

The result is a working proof of concept and shows some promising results. However, more work is needed if the solution is to be used for real-life testing of program transformation correctness.

Preface

This thesis is the result of an eight month research period, 4.5 of which were spent at Eötvös Loránd University (ELTE) in Budapest. The trip to Budapest was undertaken together with fellow student Ely Deckers, whose work I will reference in this thesis.

I would like to thank the following people:

- Zoltán Horváth and László Lövei at the Department of Programming Languages and Compilers of ELTE for their support during my research project;
- István Bozó and Melinda Tóth for answering all my questions about Erlang, RefactorErl and QuickCheck;
- Rinus Plasmeijer for being my supervisor and for his continuous feedback during the writing of this thesis;
- Pieter Koopman for being the second reader of my work;
- Bálint Fügi for arranging excellent accommodation;
- Emil Megyesi for making me feel at home in Budapest;
- Ely Deckers for being a good research partner and for making my time in Hungary most memorable.

Contents

1	Introduction	5
1.1	The Erlang programming language	5
1.2	What is refactoring?	6
1.3	RefactorErl: a refactoring tool for Erlang	6
1.4	Testing refactoring transformations	8
1.4.1	Past results	8
1.4.2	General approach	9
1.5	RefactorErl's semantic graph	11
2	QuickCheck	13
2.1	Introduction	13
2.2	Examples	13
2.3	Using QuickCheck to test refactorings	14
3	Testing the Rename Function refactoring	16
3.1	Properties to be tested	16
3.2	Implementation	17
3.3	Test results	19
3.4	Limitations of the semantic graph	19
4	Expressing semantics by symbolic computation	21
4.1	Introduction	21
4.2	Area of applicability	23
4.3	Introductory example	24

4.4	Types	25
4.5	Symbolic representation	26
4.6	The symbol table	27
4.7	Handling uncertainty	27
4.8	Evaluating case expressions	28
4.9	Evaluating function applications	30
4.10	Canonization	32
4.11	Extended examples	33
	4.11.1 Function application: The area of a circle	33
	4.11.2 Uncertainty: Absolute value	34
5	Analyzing correctness of refactoring transformations	36
5.1	Introduction	36
5.2	Theoretical properties	36
5.3	Empirical results	38
5.4	Eliminate variable	38
5.5	Rename function / Rename variable	41
6	Discussion	43
6.1	Runtime errors	43
6.2	Side effects	44
6.3	Explosion of possibilities	44
7	Conclusions and recommendations	45
7.1	Introduction	45
7.2	Personal thoughts	45
7.3	Strong points	46
7.4	Weak points and future research	46

Chapter 1

Introduction

1.1 The Erlang programming language

Erlang is a functional programming language developed by Ericsson. It was originally designed to handle telecommunication processes, but nowadays it is being used in a broader field. The main differences with languages like Haskell or Clean are the following:

- Erlang has strict evaluation. All subexpressions must be fully evaluated before an expression is evaluated. Lazy evaluation can be achieved, but it is not a native feature of the language.
- The language is dynamically typed. This means that function arguments can have multiple types, and that the return value of a function doesn't necessarily have to be a fixed type. There is no static type checker or type inference mechanism. This leads to potential runtime errors, which are acceptable within the Erlang philosophy that system components are allowed to crash, as long as a recovery mechanism is provided.
- Algebraic Data Types are not supported.
- Currying is not supported.

In addition, Erlang supports hot swapping of program code (“hot code loading”), enabling developers to patch a system without having to interrupt its execution. Finally, Erlang was especially designed for handling massive concurrency. For more details on the design decisions taken during the development of Erlang, see [1].

1.2 What is refactoring?

Refactoring is “[...] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” [2] Refactoring tools exist for virtually every mainstream programming language, and are often integrated in the IDE. For example, Eclipse [3] and Visual Studio [4] provide built-in refactoring functionality. A separate refactoring tool for Haskell, called HaRe, is in development [5].

Perhaps the simplest example of refactoring is renaming a variable. Suppose that the following Java code fragment has been loaded into Eclipse:

```
1 for(int i = 0; i < size; i++) {
2     System.out.println("Current number: " + i);
3 }
```

Snippet 1.1: Java example (original)

The developer can now right click a variable name, for example `i` in the fragment above, select “Refactoring” → “Rename...” and enter a new variable name, for example `counter`. The refactoring algorithm scans the source code for occurrences of variable `i` and replaces every instance with its new name. The fragment now reads:

```
1 for(int counter = 0; counter < size; counter++) {
2     System.out.println("Current number: " + counter);
3 }
```

Snippet 1.2: Java example (refactored)

While performing the above refactoring it is essential that certain conditions are taken into account. The refactoring algorithm has to be aware of the fact that `i` is only visible within the scope of the `for`-loop. Other variables named `i` that might be present in the source code should be left unchanged. This is one of the many conditions that a refactoring algorithm has to take into account, and these conditions vary for every refactoring transformation.

1.3 RefactorErl: a refactoring tool for Erlang

Since 2006 the Department of Programming Languages and Compilers at Eötvös Loránd University has been researching the possibilities of Erlang refactoring, in collaboration with Ericsson. At the time of writing, 22 refactoring transformations (including renaming transformations) have successfully been implemented in a separate refactoring tool, called RefactorErl [6]. A similar research project is being carried out by a team at the University of Kent. Their refactoring tool Wrangler [7] provides functionality similar to RefactorErl. There is a key difference between both projects. Wrangler performs refactoring by traversing the parsed syntax tree repeatedly for each refactoring condition, and then once more for the refactoring transformation itself, thereby parsing the code multiple times. RefactorErl maintains semantic information in order to avoid multiple

tree traversals. There has been an attempt to merge the two tools, but these efforts have failed [8].

In order for an Erlang module to be refactored with RefactorErl, it has to be loaded into the RefactorErl database. While loading a module, the RefactorErl lexer and parser analyze the module and create a semantic graph based on the code. Basically, this is an Abstract Syntax Tree with some extra edges to facilitate quick searching. Once a refactoring step is invoked, the semantic graph is analyzed and transformed into an updated version that represents the refactored source. RefactorErl then generates new source code based on the updated graph and shows it to the user. RefactorErl has been integrated with Emacs to provide the user with a friendly interface.

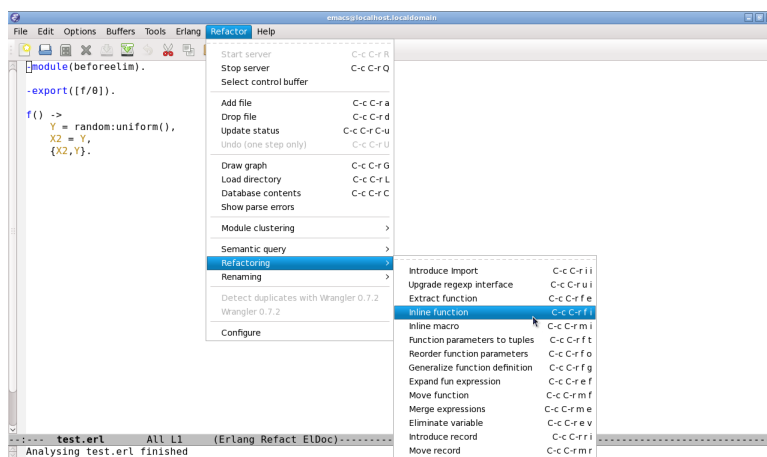


Figure 1.1: A screenshot showing the Emacs interface for RefactorErl

The following refactoring transformations are supported by the tool:

- | | |
|--------------------------------|---------------------|
| Introduce export | Eliminate variable |
| Upgrade regexp interface | Introduce record |
| Extract function | Move record |
| Inline function | Move macro |
| Inline macro | Rename module |
| Function parameters to tuples | Rename header |
| Reorder function arguments | Rename function |
| Generalize function definition | Rename variable |
| Expand fun expression | Rename record |
| Move function | Rename record field |
| Merge expressions | Rename macro |

For a detailed explanation of each of the refactoring steps, see [9].

1.4 Testing refactoring transformations

1.4.1 Past results

When performing a refactoring transformation on a piece of source code, one must be aware of the conditions under which the transformation may take place. In this field, these are called *side conditions*¹. For example, it is not allowed to rename a function if the renamed function conflicts with an existing function with the same name and arity. Using case studies the RefactorErl research group has defined lists of side conditions for each refactoring step.

For the *rename function* refactoring, the following case studies (each detailing a specific circumstance) were defined [9]:

- Multiple function clauses
- Static calls
- Implicit fun expression
- Exported function
- Imported function
- Name clash with a local function
- Name clash with an imported function
- Name clash with an auto-imported built-in function
- Name clash with a local function in a module that imports the function to be renamed
- Name clash with an imported function in a module that imports the function to be renamed

For each of these cases the developers came up with examples and discussed whether the *rename function* refactoring would be valid in that case and if so, what the refactored code would look like. These discussions were the basis for evaluating the conditions for each refactoring transformation when implementing the algorithms. Implementing these conditions in the refactoring tool should theoretically prevent the refactoring algorithm from generating incorrect source code (that is: syntactically incorrect code or code that is semantically different from the original code). In practice however, the algorithms need to be tested.

In order to verify correctness a number of testing techniques were used. Automated differential testing was done by using a script that compares RefactorErl

¹One could argue that *pre condition* would be a better term than *side condition*. When asked about this, László Lövei stated “Probably it would be more appropriate, but ‘side condition’ is the usual term used in this context. Maybe because refactorings can usually be executed even when their side conditions are not met, but in that case they are not preserving behaviour.” [8]

results with the results of its counterpart Wrangler. More precisely, a random set of transformations and parameters were generated for a discrete set of test subjects. These were given to both RefactorErl and Wrangler in order to compare the results [10]. By doing so a number of defects were found, both in RefactorErl and Wrangler. Evaluating this approach, it was written that the used technique “[...] involves generating such test cases by hand beforehand in the form of unit tests and refactoring regression tests, which must detail the exact operations to commit and the expected result for each case. Proper differential testing can be much wider and more consistent in scope because of the automated generation of refactoring instructions and the substitution of hand-crafted expected results by the output of one another.” [10].

Second, there is a framework that enables regression testing of refactoring transformations. It can be used to verify that no defects are introduced when a new version of the software is released. Unfortunately, this tool is undocumented at the time of writing.

Third, some experiments were carried out that automated the generation of test cases using QuickCheck (more about this in the next chapter). This approach satisfies the aforementioned desire for automated test case generation. Some properties of the *rename variable* and *rename module* refactorings were tested with random refactoring parameters generated by QuickCheck [11]. These efforts proved to be an interesting basis for this research. See section 2.3 for more information on how QuickCheck was used to test refactorings.

Parallel to this research, fellow researcher Ely Deckers developed an approach to test refactorings by means of model based testing. First, the specifications for a number of refactoring transformations (originally written in natural language) were formalized using the Z-notation. From these formal specifications QuickCheck tests were derived. Testing RefactorErl with these tests revealed some bugs, so initial results are promising. However, in order for the tool to reach its full potential it must be expanded so that it can examine every layer of RefactorErl more thoroughly [12].

1.4.2 General approach

In this thesis, we are looking for a method with which we can test the correctness of refactorings as precisely as possible. Ideally we would like to have an algorithm that can decide whether two programs (an original program and a refactored one) are semantically equivalent in the sense that they perform the same operations on their input. Formally, a refactoring transformation is correct if the following property holds for every program p and every transformation τ :

$$p \equiv_{sem} \tau(p)$$

In the above equation, \equiv_{sem} denotes semantic equivalence. However, semantic equivalence of programs has been proven to be an undecidable problem. The only way that we can come close to “proving” semantic equivalence is by either considering simple cases for which the problem might be decidable, or by approximation.

One method to analyze semantics is to execute a function with some example parameters and watch the outcome. If another function produces the same result with the same input, it could be equivalent for that input only. It becomes interesting when functions are executed with large amounts of inputs. It would make sense to use QuickCheck for generation of input parameters, however there are a number of reasons that make this approach questionable:

1. Erlang is a dynamically typed language. When the expected type of a function parameter cannot be inferred, random parameter generation is bound to cause a lot of runtime errors. Even when comparing two functions including the possibility of runtime errors, the process of randomly generating test data is quite pointless since there is no guarantee that *relevant* parameters are being generated.
2. Executing a single function is impractical because in real life programs, functions are always called in an environment. One could provide a framework that builds this environment for the function under test. However, this environment is specific for each function and there is no generic way of building a suitable environment.
3. Since Erlang's main application area is server processes, many programs consist of functions that execute in an endless loop. The presence of non-terminating functions makes testing by execution impractical.
4. A function might have side effects, causing it to produce different results with consecutive calls.

Taking the arguments above into consideration, we chose another approach to the problem. A first attempt checks several formal properties of RefactorErl's semantic graph that should hold for before and after performing a semi-random refactoring with QuickCheck. The results are described in chapter 3. Although powerful, we felt that it lacked a certain precision. For an alternative approach we created a static evaluator that approximates program semantics as well as possible. The resulting program representations can be compared and semantic equivalence (as approximated by the algorithm) can be decided upon. These results are described in chapters 4 and 5. In chapters 6 and 7 a discussion of the solution and conclusions and recommendations are given.

1.5 RefactorErl's semantic graph

The central datastructure that is used by the refactoring tool is the semantic graph. It is produced by the RefactorErl parser that parses modules loaded into the RefactorErl database. Figure 1.2 shows a partial semantic graph for the Erlang function $f(X) \rightarrow X * X$.

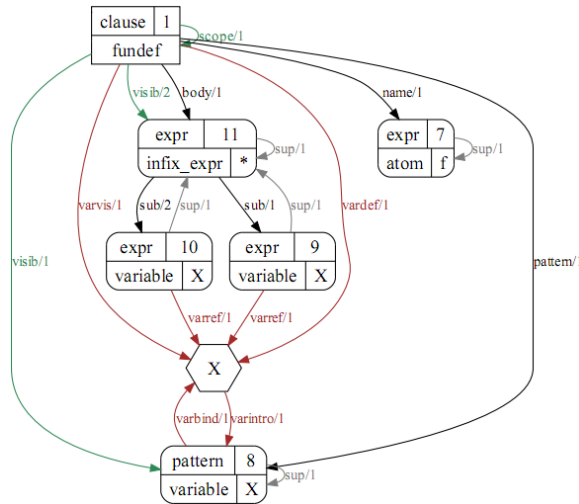


Figure 1.2: Example semantic graph

The boxes represent the syntactic nodes, hierarchically connected by black arrows, while the hexagons represent semantic nodes. Colored arrows represent visibility information: the *fundef*-node defines variable X , while other nodes refer to it. As modules grow this graph can become very complex, with dozens of nodes and hundreds of references between them.

To allow for easy traversal of the graph, the research team has implemented an interface that offers a number of graph querying functions. Starting at the root node that is the starting point of every graph (not shown in figure 1.2), these functions traverse the graph and filter out the requested information. Using a chain of query functions, one can for example find the node corresponding to a specific function, and then request all its variables. This information can then be used by the refactoring algorithm. It can analyze the structure of the program and call query functions that update the graph to transform it into a new state, representing the refactored code. Alternatively the query functions

can be used to verify certain properties that must hold for every graph, in order to check that it is a correct representation of a program. An example property: *all variables used in a module must correspond to an existing semantic node*. If not, it means that there are variables which are unknown in the scope in which they are referenced². This property can be used in combination with the refactoring transformation *rename variable*, for instance. The property should hold before and after refactoring.

It is trivial to perform this analysis on a single instance of a graph, belonging to one program. However, it becomes more interesting when one is able to automatically generate many different refactoring outcomes and thereby many different graphs. The defined property (among others) must hold for *all* generated test cases. The following chapter describes what QuickCheck is and how it can be used to generate these test cases in order to test refactorings.

²Such code wouldn't even compile, but it can occur that an incorrect implementation of a refactoring transformation creates a graph in which there are unbound variables. This can be detected by testing.

Chapter 2

QuickCheck

2.1 Introduction

QuickCheck is “[...] a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases.” [13] Test case generation is done randomly, by putting “[...] distribution [of test cases] under the human tester’s control, by defining a *test data generation language* [...], and a way to observe the distribution of test cases.” [13] QuickCheck was originally designed for Haskell, although adaptations for other languages, including Erlang, exist [14].

The best way to understand QuickCheck is by looking at some examples.

2.2 Examples

Suppose that the following simple stack-related functions are implemented in Erlang:

```
1 stack_push(X, Stack)    -> [X|Stack].
2
3 stack_pop([])           -> cannot_pop_empty_stack;
4 stack_pop(_|XS)        -> XS.
```

Snippet 2.1: Stack functions

Note that `X` and `Stack` in `stack_push/2` match all types. This results in a possibly mixed stack.

A useful property for testing could be: for all possible stacks of type `[Int]`, adding another element and then removing it again must yield the original stack. Formally:

$$\forall \text{Stack} : [\text{Int}] \forall X : \text{Int} . \text{stack_pop}(\text{stack_push}(X, \text{Stack})) = \text{Stack}$$

The QuickCheck notation is very similar to the mathematical notation:

```
1 prop_pushpop() ->
2   ?FORALL(Stack, list(int())),
3     ?FORALL(X, int()),
4       stack_pop(stack_push(X, Stack)) == Stack).
```

Snippet 2.2: QuickCheck property

After calling the aforementioned property by entering

```
eqc:quickcheck(my_stack:prop_pushpop())
```

QuickCheck responds with:

```
OK, passed 100 tests
true
```

This means that for 100 randomly generated test cases, no counter-example was found.

To guide the generation of test cases, it is possible to define custom generators. The following function generates lists of integers that are even numbers:

```
1 even_int_list() ->
2   ?LET(I, list(int()),
3     lists:filter(fun(X) -> X rem 2 == 0 end, I)).
```

Snippet 2.3: Custom generator

?LET is a binding operator that binds `I` to the list generated by `list(int())` and uses it in the second generator: `lists:filter(...)`.

The generation of test cases is done in a random manner, and by default in a strict way. However, if the generation of test cases is a costly operation one can use lazy generation. This is done by using the `?LAZY(G)` generator that takes another generator `G` as an argument, but generates its data only when it is needed.

2.3 Using QuickCheck to test refactorings

The research team has done some experiments with QuickCheck as a tool to test properties of refactoring transformations. The tested refactorings are *rename module* and *rename variable*. Normally, when a user performs the *rename variable* refactoring, he highlights a variable in the Emacs editor, chooses the refactoring in the menu and provides a new variable name. Thus, the refactoring algorithm needs the following arguments: (1) a pointer to a file on the disk, (2) a character position in that file and (3) a new variable name. With QuickCheck generators, it is easy to generate a large amount of these arguments automatically.

With a predefined set of modules loaded into the RefactorErl database, the testing module does the following:

1. It randomly selects one of the files in the database (this is argument 1);
2. From this file, it extracts all variables;
3. For each variable, it requests its corresponding character position in the file;
4. From the list of character positions, it selects one randomly (this is argument 2);
5. It calls a custom generator that randomly generates a fresh variable name, in accordance with the Erlang syntax for variables (this is argument 3);
6. It calls the *rename variable* refactoring function with the generated arguments.

Functionality such as extracting variables is done by querying the semantic graph. After performing the refactoring on the source code, a number of formal properties are checked. These properties must hold before and after refactoring, to ensure that the algorithm does not break them. The properties are:

1. If the code compiles before refactoring, it should also compile after refactoring;
2. The binding and reference structure of the variable must be the same before and after refactoring;
3. The binding and reference structure after renaming to the original name must result in the same binding and reference structure as the base state.

If all properties hold the algorithm continues with the next test case until a predefined number of cases have been tested. If a counterexample is found for which one or more of the defined properties do not hold, it might indicate that the refactoring algorithm is not 100% correct. On the other hand, if no counterexample is found within the limited amount of test cases, that does not mean that the refactoring is entirely correct.

In summary, QuickCheck's method of testing provides partial decidability concerning the correctness of the program under test. A counterexample indicates a possible error, but the absence of counterexamples only builds confidence instead of proving correctness.

Chapter 3

Testing the Rename Function refactoring

3.1 Properties to be tested

One of the existing refactoring transformations is concerned with renaming a function. As with every refactoring, there are a number of side conditions that should be checked. Only if the transformation meets these conditions, the refactoring is allowed. An example side condition for the *rename function* refactoring is that the new function name should not already exist for a function with the same arity.

The *rename function* refactoring should consider all language constructs that are affected by renaming a function. For example, if the renamed function is imported in another module its reference in this module should also be updated. More precisely, we defined five different classes of semantic nodes that can be connected to a function node in the semantic graph. These are:

1. The definitions of the function;
2. The applications of the function;
3. The occurrence of the function in the export list;
4. Implicit references to the function (when a reference to the function is used as a function argument for a higher order function);
5. The places where the function is imported.

For the scope of this discussion, the total of all these nodes will be called the function's *semantic context*. When a function is renamed it is important that its semantic context remains unchanged¹. This means that the semantic nodes

¹The nodes in the semantic graph are identified by RefactorErl by assigning auto-incrementing IDs to them. This causes the graph to stay the same when a property of a node, such as its name, is changed.

that were connected to the renamed function node are still connected to it after renaming. If not, this means that the refactoring transformation has performed an illegal operation, or was not thorough enough in updating all necessary parts of the program.

We used QuickCheck to perform many random *rename function* refactorings on a static set of loaded modules. First, all names of the existing functions were extracted. Second, by querying the semantic graph and the lexical nodes that are part of it, the positions in the files at which these function names occur were retrieved. QuickCheck then generated a list of random function names, in which it included existing function names as well. The latter was done to cover the border test cases, namely the instances in which a function is renamed to an existing function. RefactorErl should disallow this refactoring, and thus it is behavior that can be tested. From the list of file positions and generated function names, one pair was chosen at random and these were the parameters with which the refactoring was called.

While generating these refactorings, there were a number of properties that should hold for the old and the new situation. We tested three properties:

1. If the code compiles before refactoring, it should also compile after refactoring;
2. The semantic context for the function node should remain unchanged;
3. The arity of the function should remain unchanged.

3.2 Implementation

The following code listing contains the top-level function that generates refactoring arguments, performs the refactoring and tests the properties.

```

1 prop_rename_fun() ->
2   Files = get_files_in_database(),
3
4   ?FORALL(Args, ?LAZY(generate_args(Files)),
5   begin
6     FunNode = ?Args:function(Args),
7     ModNode = ?Args:module(Args),
8     NewName = ?Args:name(Args),
9
10    [FileNode] = ?Query:exec(ModNode,?Mod:file()),
11    FilePath   = ?File:path(FileNode),
12
13    ArityBefore = ?Fun:arity(FunNode),
14
15    ContextBefore = lists:flatten(
16      [get_context(Fun) || Fun <- ?QCCOMMONFUN:get_funs(
17        Files)]),
18
19    CompResultBefore = compile:file(
20      FilePath, [strong_validation, return_errors]),
21
22    Result = ?QCCOMMON:perform_transformation(
23      reftr_rename_fun, Args),
24
25    case Result of
26      {result, _} ->
27        prop_compilation(FilePath, CompResultBefore) andalso
28        prop_context(Files, ContextBefore) andalso
29        prop_arity(ModNode, NewName, ArityBefore)
30      _ ->
31        true
32    end
33  end).

```

Snippet 3.1: Erlang implementation of *rename function* testing module

This module uses a custom QuickCheck generator (line 4) to generate refactoring arguments. The function `generate_args/1` returns a set consisting of a random function node pointer, a module node pointer and a string representing the new function name. Using this information the corresponding file node pointer is requested (line 10), and using the file pointer the path to the file in the filesystem can be found (line 11). Three properties are stored for later comparison: the original arity of the function, its semantic context and its compilation result. Then, the function is refactored (line 21). If the refactoring was allowed, the refactoring function returns a record `{result, ...}`. Three QuickCheck properties are then checked: (1) the module's compilation result should be the same, (2) the semantic context should be the same and (3) the function arity should be the same. Each property requests the up-to-date information about the function and compares it with the values originally stored. If the property holds, it returns true. If all properties hold, the QuickCheck generator continues with the next test case until a predefined number of tests have been run.

Learning how to work with QuickCheck, how to interpret the semantic graph and how to use the query functions took quite some time during the early stages of our research. However, once we were comfortable with these systems the testing module demonstrated here was quickly written.

3.3 Test results

Fellow researcher Ely Deckers further developed the aforementioned script. While testing a number of modules with it he found two types of errors in the *rename function* refactoring [12]:

1. In an early version of the Erlang platform, an alternative way of calling a function was supported. Normally, one would use `module:function(arg)`, but the following was also valid: `{module, function}(arg)`. The *rename function* refactoring does not take the alternative syntax for function calls into account and fails to refactor them. However, in the RefactorErl documentation it is stated that the developers deliberately chose *not* to support these alternative function calls, since they are deprecated.
2. At one point the whole RefactorErl tool was refactored itself. Many things were renamed and some modules were redesigned. During this process, there was a build of the tool in which the *rename function* refactoring was broken. If one renamed, for example, a function `f` to `g`, and then back to `f`, the refactoring would break the program. This type of error cannot exist long without being detected by users, so it was fixed quickly. Nevertheless, the testing module found the bug as well.

In conclusion, the testing module found no “real” errors in the refactoring algorithm, but its capability to detect the errors mentioned above proves its validity.

3.4 Limitations of the semantic graph

Although the semantic graph and the query language associated with it are very powerful tools, they are unable to detect certain types of incorrect refactorings. It could occur that a refactoring is correct as far as graph consistency is concerned, but the semantics (in terms of a function’s return value) could change without being detected.

Consider the following example of the *eliminate variable* refactoring:

```
1 f() ->
2   X = 1 + 2,
3   X * X.
```

Snippet 3.2: Original function

```
1 f() ->
2   (1 + 2) * (1 + 2).
```

Snippet 3.3: Refactored function

In this example, the variable `X` is eliminated and replaced by its value. Using the semantic graph, perhaps the most thorough analysis one could perform is to check whether the variable bindings of the other variables in this function (in this case none) remain unchanged. But there are many things that can go wrong, which cannot be detected in this way. What if the parentheses were omitted and the expression would read `1 + 2 * 1 + 2`? The function's semantics would drastically change.

In order to analyze a function's semantics, we defined a new data structure that uses static computation to describe a function's inner workings.

Chapter 4

Expressing semantics by symbolic computation

4.1 Introduction

While the semantic graph contains information about values, types and mathematical operators, the query language is not capable of analyzing computation. It is desirable to have a method of examining function semantics, to be used by current and future tests of refactoring transformations. If one can prove that a function's semantics remains unchanged after refactoring, one can be sure that the refactoring was correct in that case. We already stated that it is generally undecidable whether two functions are semantically equivalent. We therefore followed another approach that can *estimate* semantic equivalence, or decide upon it in a limited number of cases. For this purpose we developed a specific mechanism that can express semantics of Erlang functions. Ideally we would like to be able to evaluate the whole Erlang language. This is a very extensive task, and moreover it is impossible to decide for every Erlang function whether it is equivalent to another function. Therefore the aim is to prove semantic equivalence of relatively simple functions, for instance the one described in snippet 3.2.

Existing tools that use static evaluation in this context include TypEr [15] and Dialyzer [16], two closely related tools that attempt to infer type information for Erlang code. Their aim is to analyze Erlang programs for potential errors caused by type-related programming mistakes. Being able to infer type information can be helpful when comparing expressions, but the type information that these tools generate is not specific enough for the purpose of this research.

We built an evaluator that traverses the semantic graph and tries to statically compute what a function's return value will be. Evaluating a function results in a tree structure in which each node represents a certain component of the language, for example an operator or a function application. Using the evaluator, one can prove correctness of refactoring transformations (for that specific case)

by first evaluating the function to be refactored, then refactoring the function, and then evaluating it again. If both evaluation results are equivalent, it means that the refactoring did not change the semantic meaning of the function and therefore it was correct. Erlang’s built-in operator `==` is overloaded for tree structures (being nested tuples) and this equivalence relation is used to decide whether the trees are equivalent. Figure 4.1 shows the approach used by the evaluator.

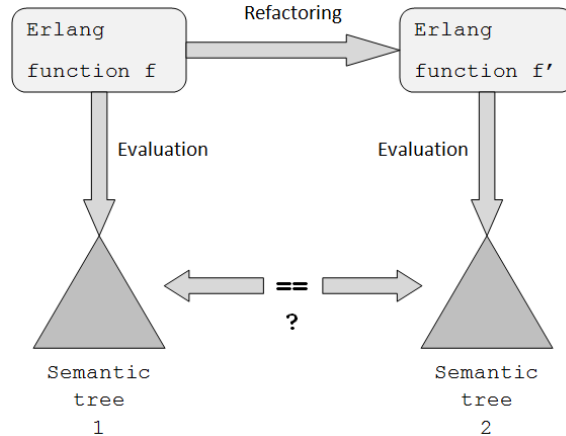


Figure 4.1: Determining semantic equivalence by evaluation

This approach creates the following property for Erlang functions:

$$\text{eval}(f) \equiv_{syn} \text{eval}(f') \Rightarrow f \equiv_{sem} f'$$

This means that syntactic equivalence of semantic trees, obtained by evaluating both functions f and f' , implies semantic equivalence of these functions. In this property, f' represents the refactored version of f , so $f' = \tau(f)$.

Note that this property is only true for the subset of Erlang for which the evaluator has been implemented. In general, the reverse property (semantic equivalence of functions implies syntactic equivalence of trees) does not hold. A simple counterexample is the expression $X + 1$ versus $1 + X$. One could claim that these expressions are semantically identical, but since properties like commutativity of operators have not been implemented, these two expressions remain different as far as their semantic trees are concerned. If needed, it would require relatively little effort to implement a number of arithmetic rules.

4.2 Area of applicability

The ideal situation would be to cover the entire Erlang language in the symbolic evaluator, but due to the vastness of this task only a subset of the language can be statically evaluated. The following BNF grammar describes exactly which subset of Erlang can be evaluated by the tool.

```

ProgramForms := FunctionDeclaration+
FunctionDeclaration := FunctionClause+
FunctionClause := FunctionSymbol FunClause
FunctionSymbol := AtomLiteral
FunClause := (Pattern*)Body
Pattern := AtomicLiteral | Variable | UniversalPattern
AtomicLiteral := IntegerLiteral | FloatLiteral | CharLiteral |
  StringLiteral | AtomLiteral
UniversalPattern := -
Body := Expr+
Expr := MatchExpr | InfixExpr | PrefixExpr |
  ApplicationExpr | PrimaryExpr
MatchExpr := Pattern = InfixExpr
InfixExpr := PrefixExpr InfixOp PrefixExpr
InfixOp := + | - | * | / | == | /= | < | <= | > | >=
PrefixExpr := PrefixOp ApplicationExpr
PrefixOp := + | -
ApplicationExpr := FunctionSymbol (Expr*)
PrimaryExpr := Variable | AtomicLiteral | CaseExpr |
  ParenthesizedExpr
CaseExpr := case Expr of CrClause+ end
CrClause := Pattern Body
ParenthesizedExpr := (Expr)
IntegerLiteral
FloatLiteral
CharLiteral
StringLiteral
AtomLiteral
Variable

```

This grammar is based on the official grammar for the Erlang language [17]. The last few production rules are trivial and therefore not defined in the above grammar to favor readability. They correspond to the notation for Erlang integers, floats, chars, string, atoms and variables respectively. Formally, in addition to a number of functions, an Erlang module also contains some compiler directives,

a module name, etcetera. These are ignored by the tool and thus left out of the grammar.

Apart from restrictions on the language, the tool has only been tested with small Erlang programs. A typical real-life program is large and complex; when attempting to statically evaluate such a program it is likely to result in a huge number of possible outcomes. As the number of possibilities grows the tool might become slow. See also 6.3.

4.3 Introductory example

Every Erlang function is basically a collection of statements, which are executed in order, one after the other. Typical operations within a function include instantiation of one or more variables, function applications, evaluation of one or more conditions by means of a `case` or `if` expression, and eventually returning the function's result.

The symbolic evaluator analyzes function statements one by one, starting with the first. Whenever a variable is instantiated, its value is stored in a symbol table that is maintained within the scope of the function. If at a later time that variable is used, the evaluator performs a table lookup and replaces the variable instance with its concrete value, if such a mapping exists. By replacing variables with their values, a more concrete representation of the function's semantics can be given.

Before discussing the evaluator in detail, a simple example is given to illustrate its functionality.

```
1 f () ->
2   X = 4 - 1,
3   X.
```

Snippet 4.1: Symbolic computation example

The first expression, `X = 4 - 1`, is a so-called *match expression* with an *infix expression* on the right hand side. The match expression assigns a value to a variable by means of the `=` operator. The infix expression consists of two constants, bound together with a `-` operator. The infix expression is represented as follows:

```
{infix_expr,
 {
  '-',
  {constant,{integer,4}},
  {constant,{integer,1}}
 }
}
```

However, since the computation holds no unknown values, its result will immediately be calculated. Thus, the entire first expression will be:

```

{match_expr,
 {
  {variable,{v0}}, [{constant,{integer,3}}]
 }
}

```

The symbol table is updated by adding variable `X` with value `{constant,{integer,3}}`. The evaluator abstracts from the concrete variable name `X`, and substitutes an internal representation `v0` in its place.

The second line of the example function returns the value for `X`. By performing a lookup, the evaluator sees that the value for `X` equals `[{constant,{integer,3}}]` and this will be the function's return value.

In the following sections every aspect of the evaluator is described in detail.

4.4 Types

The evaluator function has the following type:

```

sc :: FunctionName FunctionArity ClauseNr State ->
    {SymbolicRepresentation, State}

```

`FunctionName` and `FunctionArity` indicate which function needs to be analyzed. `ClauseNr` is a number that indicates which function clause needs to be called. This has to be made explicit because in case of function arguments that are unknown, it is impossible to know which function clause will match. `State` is a record type, which keeps track of three variables that need to be available at all times. These are:

SymbolTable

The symbol table in which variable values are stored. See section 4.6.

OptionList

A list of tuples in which a number of options can be stored to change the program's behavior.

VarCounter

A simple counter that holds the last variable name used for the internal representation of variables. See section 4.6.

For the `OptionList`, the following options are supported:

call_depth

The depth with which function calls are evaluated. See section 4.9.

`abstr_fun_names`

Set to `true` if the function names should be omitted in `application` nodes. See section 4.9.

`canonical`

Set to `true` if the resulting semantic tree should be canonized. See section 4.10.

4.5 Symbolic representation

The result of the function, `SymbolicRepresentation`, is a tree structure that describes the function's semantics. This tree structure can consist of the following nodes:

`{expr_seq, {[ExprList]}}`

A list of expressions; the expressions in `ExprList` are executed in order.

`{expr_choice, {[ExprList]}}`

A choice of expressions; one of the expressions in `ExprList` will be executed.

`{application, {FunName, [Arg1, ..., Argn]}}`

A function application. If `abstr_fun_names` is set to `true`, `FunName` will read 'function'. See section 4.9 for more details.

`{match_expr, {LhsExpr, ValueList}}`

An expression of the form `LhsExpr = ValueList`. This can be a variable assignment or a pattern match. Variables can have multiple possible values, that is why a `ValueList` is used. At the moment, only variable assignments are supported.¹ See section 4.7 for more details.

`{infix_expr, {Operator, ValueListLhs, ValueListRhs}}`

An expression with an infix operator, for example `a + sign`.

`{constant, {atom, Value}}`

`{constant, {boolean, Value}}`

`{constant, {char, Value}}`

`{constant, {float, Value}}`

`{constant, {integer, Value}}`

¹In fact, Erlang doesn't have "variable assignment" at all. Expressions of the form `X = i` are called match expressions since they match `X` with `i`. For example, `X = 1`, followed by `X = 1` is accepted. However, if the second statement reads `X = 2` an error is produced stating that `X` (already having value 1) and 2 do not match. So the real situation is slightly more subtle than how it is interpreted in this thesis.

```
{constant, {string, Value}}
```

```
{variable, {IntName}}
```

A variable with the name used for the internal representation.

4.6 The symbol table

The symbol table is an Erlang dictionary, which behaves like a hash table. The keys in this table are variable names and the values are lists of possible values for that variable. Since it is undesirable to hold on to the specific variable names used in the functions under evaluation (for example when testing the *rename variable* refactoring), the symbol table abstracts from concrete variable names. To achieve this, the symbol table consists of two parts: a part that maps every concrete variable name to its internal representation, and a part that maps every internal variable name to its value list.

When a variable is encountered for the first time, two records are added to the symbol table. One to store the relation between the variable name and a fresh internal name, and one to store the relation between the internal name and the assigned value(s). For example, the expression `X = 1` results in the following entries:

```
X    ⇨ v0
v0   ⇨ {constant, {integer, 1}}
```

The evaluator sees that it is the first occurrence of `X`, therefore it creates a mapping between it and a fresh internal variable name. The right hand side of the match expression results in `{constant, {integer, 1}}`, and that value is mapped to the internal variable name `v0`.

The symbol table provides functions for creating new internal variable names and storing values, as well as lookup functions.

4.7 Handling uncertainty

Due to the existence of branch statements depending on values that can be unknown, the notion of uncertainty is introduced in the description of function semantics. Consider the following function:

```
1 f(X) ->
2   case X of
3     1 -> true;
4     _ -> false
5   end.
```

Snippet 4.2: Uncertainty

The return value of `f/1` depends on its argument `X`. If the value of `X` is unknown it is impossible to predict which pattern of the case expression will match. In

the general case, the evaluator solves this by enumerating all possible outcomes in an `expr_choice` node. The semantic representation for `f/1` will be:

```
{expr_choice,
  [{constant, {boolean, true}},
   {constant, {boolean, false}}]}
```

This is the reason why all variables in the symbol table have a list of *possible* values, instead of just a fixed value. This notion of uncertainty has an impact on mathematical calculation. For all implemented mathematical operators with arity 2, the operator works on lists of possible values, producing a list of all possible results. Formally:

$$P \odot Q = \{p \odot q | p \in P, q \in Q\}$$

Where \odot represents any operator, like $+$, $<$, $==$, etc. Thus, calculating $\{1, 2\} * \{3, 4\}$ will yield $\{3, 4, 6, 8\}$. Calculating $\{1, 4\} < \{2, 3\}$ will yield $\{true, false\}$.

A more detailed method of dealing with branch statements is given in the next section.

4.8 Evaluating case expressions

Case expressions in Erlang have the following structure:

```
case head_expression of
  pattern_1 -> expressionlist_1
  pattern_2 -> expressionlist_2
  ...
  pattern_n -> expressionlist_n
end
```

Patterns can be constructed to catch multiple types and/or values. The patterns `0`, `[]`, `an_atom` are all valid within the same case expression. A special pattern is the underscore `_`, which matches everything. Typically it is used for the last pattern to catch all previously uncaught cases.

For representing these case expressions, a solution could be to simply enumerate all expressions on the right hand side of the patterns in an `expr_choice`. However, when the value of the head expression is known, the patterns can be evaluated and only the possible matches are considered “solutions” for the case expression. Both the head expression and the pattern expressions can represent a list of possible values, due to the value uncertainty discussed in the previous section. A pattern potentially matches if in its value set there is at least one value that is also present in the head expression’s value set. An example (in pseudo-code):

```

1 case {1, 2, 3} of
2   {0}   -> A
3   {1, 2} -> B
4   _     -> C
5 end

```

Snippet 4.3: Case expression example 1

Here, the sets represent possible values. The first pattern $\{0\}$ will never match, because it shares no values with the values in the set of head expressions. The second pattern $\{1, 2\}$ shares two values with the head set, so it is a possible match. The third pattern $\{-\}$ matches everything, so it is a possible match as well. The “solution set” for this example will be $\{B, C\}$.

If it occurs that a pattern consists of a singleton set with a value that is present in the head set, it is certain that this value will never be caught by a pattern following it. In this case, that value can be removed from the head set. An example:

```

1 case {1, 2, 3} of
2   {1}   -> A
3   {2, 3} -> B
4   _     -> C
5 end

```

Snippet 4.4: Case expression example 2

The first pattern matches 1. There are three possible values for the head expression; if it had value 1 the resulting solution would be A, and this would conclude the case expression. This fact allows removal of the value 1 from the head set. The second pattern $\{2, 3\}$ is a possible match for the updated head set $\{2, 3\}$. Note that these sets still represent *possibilities*, so if the head expression had value 2 and the pattern had value 3, it wouldn’t be a match. This is why elimination of $\{2, 3\}$ from the head set is not allowed. Finally, the last pattern matches the remaining possibilities. The solution set for this example equals $\{A, B, C\}$.

Using this algorithm, there are two methods of termination:

1. If the head set becomes empty when there are still patterns left, it means that the remaining patterns will never match and that they can be ignored.
2. If all patterns are evaluated and the head set is not empty, it means that there might be values in the head set for which there are no patterns. When executed, this could result in a runtime error.

Formalizing these notions, the following algorithm emerges that evaluates case expressions (in pseudo-Erlang):

```

1 e([], PS, ES, S)          -> S
2 e(H, [], ES, S)         -> S ++ [{runtime_error_no_match}]
3
4 e(H, [P|PS], [E|ES], S) ->
5   P == underscore        -> S ++ E
6   intersect(H, P) != [] && length(P) == 1 -> e(H -- P, PS,
7     ES, S ++ E)
8   intersect(H, P) != [] && length(P) > 1 -> e(H, PS, ES,
9     S ++ E)
10  otherwise              -> e(H, PS, ES,
11    S)

```

Snippet 4.5: Case expression evaluation algorithm

H is the list of head expressions, PS is the list of patterns (each pattern being a list of expressions), ES is the list of expression lists on the right hand sides and S is a list that accumulates the possible expressions that are the result of the case expression, here called the solution list.

If the pattern equals the underscore symbol it always matches, so the corresponding right hand side expressions are added to the solution list and the algorithm terminates, since patterns following the underscore will never match. If the head expression and the pattern have at least one value in common (that is: their intersection is not empty) it is a possible match, so the expressions are added to the solution list. If the pattern consisted of only one value (its length equals 1), this value can be removed from the list of head expressions, since this value will never be caught by another pattern. If the head expression and pattern have no values in common, the algorithm recursively continues with the next pattern.

If the list of head expressions becomes empty, it means that for all possible values there is a corresponding pattern. Remaining patterns will never match so the algorithm terminates by returning the solution list. If, however, the patterns are all evaluated and the list of head expressions is non-empty, the algorithm terminates by returning the solution list, plus an atom indicating that there might not be a match.

There is an exception to the above algorithm: if the value list of the head expressions contains at least one unknown variable, it is undecidable which patterns will match. In this case all right hand side expressions are part of the solution set, including the `runtime_error_no_match` atom. Unfortunately this will likely be the case for real-life programs.

4.9 Evaluating function applications

When the evaluator encounters a call to another function it can do two things. If `call_depth` equals zero or the function can't be found (because it resides in another module or is a Built In Function), an `application` node is created that includes the name of the called function and a list of parameters. The expression `length(L)` would become:

```
{application, {length, [{variable, {v0}}]}}
```

A special situation occurs when an abstraction from function names is needed in the semantic representation. The option `abstr_fun_names` will be set to `true`. In this case the function names are replaced by the word “function”, so the above node will read:

```
{application, {function, [{variable, {v0}}]}}
```

This is done to be able to abstract from concrete function names, as this is desirable when testing the *rename function* refactoring, for example.

If the called function is present in the modules loaded into the RefactorErl database and `call_depth` is greater than 0, the evaluator will jump into the function and evaluate it. The function call will then be replaced by the called function’s return value. To do so, the evaluator must first find the correct clause, since functions can consist of multiple clauses. The evaluator queries all function clauses belonging to the called function and examines them one by one. A clause matches if all its arguments match the arguments in the function call. The order in which the programmer defines the function clauses is of importance, so the algorithm accepts the first clause that matches and ignores the others.

For all function calls `f(arg_1, arg_2, ..., arg_n)` and all clauses belonging to function `f/n`, the matching clause is the first one for which all arguments of the function call match the arguments in the clause definition. The function that decides whether an argument matches is the following:

```
1 arg_match(ArgCallee, ArgCaller) ->
2   case {ArgCallee, ArgCaller} of
3     {{constant, X}, {constant, Y}} ->
4       X == Y;
5     {{variable, _}, {constant, _}} ->
6       true;
7     _ ->
8       false
9   end.
```

Snippet 4.6: Function clause matching algorithm

Arguments match if and only if they are both constants and represent the same value, or if the called function expects a variable and the caller provides a constant. Constants can be atoms, booleans, chars, floats, integers and strings. More advanced data types such as lists, tuples and records are not supported, and neither are sets of possible values (see also the grammar in section 4.2). In theory, sets of possible values could be handled exactly like it is done when evaluating case expressions, but currently this has not been implemented.

To evaluate a function call, the evaluator recursively calls itself. First, a blank symbol table is created. It is then filled with entries for each function argument containing the values that are provided by the caller. The `OptionList` is reused, but the value of `call_depth` is decreased by 1. These parameters will be the starting state for the evaluation of the called function. Note that only pure

functions can be computed with this starting state (see also section 6.2). Upon completion, the evaluator returns a list of semantic trees, each corresponding to one expression, and an updated symbol table. The symbol table is discarded as it is not valid in the current scope. From the semantic tree, only the return value is kept (this will be the very last expression in the list), and the original function call is replaced by this value. An example:

```

1 f() ->
2   calc(minus, 6, 2).
3
4 calc(plus, X, Y) ->
5   X + Y;
6 calc(minus, X, Y) ->
7   X - Y.
```

Snippet 4.7: Function call example

Evaluating `f/0` with a `call_depth` greater than 0 will result in the following actions. First, all clauses for function `calc/2` are queried and they are evaluated one by one. The first clause expects the atom `plus` as the first argument, but the function call provides the atom `minus`. These arguments do not match, so the first clause will not match. The second clause expects a `minus`, so this argument matches. The second argument, `X`, matches the provided constant 6 and the same goes for `Y` and 2. The conclusion is that the second clause matches.

Then, a new symbol table is created with entries for each function argument. The first, `minus`, isn't a variable so it is skipped. The second argument, `X`, will get the value 6 and `Y` will be 2. Taking internal variable representations into account, the initial symbol table for the evaluation of function `calc/2` will be:

```

X  ↦ v0
Y  ↦ v1
v0 ↦ {constant, {integer, 6}}
v1 ↦ {constant, {integer, 2}}
```

After evaluation, the result will be:

```
{constant, {integer, 4}}
```

This value is substituted for the original function call and evaluation continues.

4.10 Canonization

Due to implementation details, it can occur that there are `expr_seq` or `expr_choice` nodes that have only one child and therefore lose their meaning. If this is the case, these nodes can safely be removed from the tree. The following small example illustrates this:

```

{expr_seq,
  {[
    {expr_choice,
      {[constant,{boolean,true}]}}
    ]}
  ]}
}

```

After canonization, this semantic tree will be:

```
{constant,{boolean,true}}
```

For the scope of this thesis, we refer to this process as *canonization*. Canonization is the process of rewriting a structure into a standard (“canonical”) form. For this purpose we define the canonical form as an instance of the semantic tree where `expr_seq` and `expr_choice` nodes have to have more than 1 child. If not, the nodes have nothing but a trivial meaning and they are eliminated.

The process of canonization can be executed to improve readability of a semantic tree. Alternatively it can happen that the evaluation algorithm produces identical semantic trees for different programs, except for trivial `expr_seq` and/or `expr_choice` nodes. We define such trees to be equivalent, but according to pure syntactic comparison they are not. In these cases we need to apply canonization.

To improve readability and to prevent the semantic trees from becoming unnecessarily complex, the examples in the following section are all given in their canonical form.

4.11 Extended examples

4.11.1 Function application: The area of a circle

The following example is a set of functions that (cumbersomely, for the sake of the example) computes the area of a circle.

```

1 area(R) ->
2   pi() * square(R).
3
4 pi() ->
5   3.14.
6
7 square(X) ->
8   X * X.

```

Snippet 4.8: Example 1

Calling `sc(area, 1, [{call_depth, 0}])` (meaning: analyze function `area/1` with call depth 0), will result in the following tree:

```

{infix_expr,
 { '*',
  {application, {pi, []}},
  {application, {square, [{variable, {v0}}]}}}
}

```

The expression `pi() * square(R)` is an infix expression with on the left hand side a function application of `pi/0` with zero arguments, and on the right hand side a function application of `square/1` with `R` as an argument. Since the call depth was set to 0, the evaluator will not try to replace these function applications by their individual return values. If the call depth is set to 1, however, the result will be:

```

{infix_expr,
 { '*',
  {constant, {float, 3.14}},
  {infix_expr, { '*', {variable, {v0}}, {variable, {v0}}}}}
}

```

The evaluator will jump into the called functions, and the call to `pi/0` is replaced by its return value 3.14. The call to `square/1` is expanded into `v0 * v0`, since the value for `R` is unknown. However, the possibility exists to call the evaluator with a non-empty symbol table. For this example, a symbol table is created which holds variable `R` with value 2. By doing so, the evaluator is able to look up the value for `R` and is able to complete the computation:

```
{constant, {float, 12.56}}
```

4.11.2 Uncertainty: Absolute value

One could say that the absolute value of every real number r can be computed by multiplying the sign of r by r itself. In Erlang:

```

1 abs(R) ->
2   sign(R) * R.
3
4 sign(X) ->
5   Positive = X >= 0,
6   case Positive of
7     true   -> 1;
8     _     -> -1
9   end.

```

Snippet 4.9: Example 2

Note that the function `sign/1` somewhat deviates from the standard definition in which `sign(0) = 0`. Evaluating function `sign/1` yields:

```

{expr_seq,
  {[match_expr,
    {
      {variable, {v1}},
      {infix_expr,
        {'>=', {variable, {v0}}, {constant, {integer, 0}}}}
    },
  expr_choice,
    {[
      {constant, {integer, 1}},
      {constant, {integer, -1}}
    ]}}}]

```

This function consists of two expressions: a match expression that assigns variable `Positive` and a case expression evaluating its result. Since the function argument `X` has already gotten internal variable name `v0` (being the first occurrence in the fresh symbol table), the variable `Positive` is represented by `v1`. The evaluated value of `v1` is an infix expression which contains `v0` as an unknown variable. Using this value as the head expression for the case construct, it is undecidable which pattern will match and thus all patterns are listed in an `expr_choice` node (see section 4.8 for a detailed explanation).

Evaluating function `abs/1` with `call_depth` ≥ 1 results in the following representation:

```

{infix_expr,
  {'*',
    [{constant, {integer, 1}}, {constant, {integer, -1}}],
    [{variable, {v0}}]}}

```

This means that the result of `abs/1` is a multiplication of the function parameter by either 1 or -1.

Following the previous example, we can supply the evaluator with a non-empty symbol table. Doing so, the variable `Positive` will get the value `true` or `false`, depending on the value of `X`, and as a result the case expression can be fully evaluated. Evaluating `abs/1` using a symbol table in which `R` equals -42, the result is:

```

{constant, {integer, 42}}

```

Chapter 5

Analyzing correctness of refactoring transformations

5.1 Introduction

In this chapter we present some of the results. First, we discuss the theoretical properties of the semantic evaluator, in comparison to properties of testing by execution. Second, we show the results that we gathered by executing the evaluator on a number of real examples.

5.2 Theoretical properties

In general, when testing functions by execution, there are a number of properties that will hold. Let p denote a program and τ a program transformation. Let $TESTS$ be the universe of all possible test cases. Then, $p \equiv_{test} q$ denotes that $\forall_{test \in TESTS} [p(test) = q(test)]$. Finally, let \equiv_{sem} denote semantic equivalence. Then:

$$\begin{aligned} p \equiv_{sem} \tau(p) &\Rightarrow p \equiv_{test} \tau(p) \text{ and thus} \\ p \not\equiv_{test} \tau(p) &\Rightarrow p \not\equiv_{sem} \tau(p) \end{aligned}$$

Furthermore:

$$p \not\equiv_{sem} \tau(p) \Rightarrow \exists_{test \in TESTS} [p(test) \neq \tau(p)(test)]$$

This means that if a test case fails, the transformation did not preserve semantics. The other way around, if the transformation is semantically correct, all test cases should pass. Finally, if we know that a transformation does not preserve semantics, there exists a test case that fails for this transformation.

Let $\text{eval}(p)$ be the resulting semantic tree after evaluating program p . The ideal solution would have the following property:

$$\text{eval}(p) \equiv_{syn} \text{eval}(\tau(p)) \Leftrightarrow p \equiv_{sem} \tau(p)$$

This property cannot be satisfied by the current implementation. We will first discuss the “left-to-right” implication:

$$\begin{aligned} \text{eval}(p) \equiv_{syn} \text{eval}(\tau(p)) &\Rightarrow p \equiv_{sem} \tau(p) \text{ and thus} \\ p \not\equiv_{sem} \tau(p) &\Rightarrow \text{eval}(p) \not\equiv_{syn} \text{eval}(\tau(p)) \end{aligned}$$

This property holds for the subset of Erlang defined in this thesis. The fact that the evaluator is only able to handle this subset prevents us from having the above property for the more general case in which programs use the entire language. In the current implementation it is easy to write functions f and g that are semantically different, but for which the evaluator produces the same trees since it skips language constructs that it doesn’t understand.

At this point it is also important to clarify that the above property only holds for trees in which function names are being used. In the case of an abstraction, the situation occurs that two trees in which the function application nodes read “function” are considered to be equivalent (given that the rest of the structure is also equivalent). In reality these can be two different functions. This is a weakness of the implementation and it causes some trees to be equivalent, while the corresponding programs are semantically different, thus violating the above property.

The “right-to-left” implication

$$\text{eval}(p) \equiv_{syn} \text{eval}(\tau(p)) \Leftarrow p \equiv_{sem} \tau(p)$$

is generally not true. For this property to hold, not only must the evaluator be extended with knowledge of the whole Erlang language, but also with a number of arithmetic rules. An example was already given in section 4.1, where it was shown that expressions $X + 1$ and $1 + X$ produce different semantic trees, even though they are semantically equivalent.

In conclusion, even though the evaluator is unable to achieve syntactic equivalence *if and only if* the programs are semantically equivalent, we can still be sure that different programs produce different semantic trees, provided that we don’t use unsupported language constructs or function name abstractions. More work is needed if we wanted the evaluator to perform better in the cases where the semantics of the evaluated functions are the same, but different trees are being produced.

5.3 Empirical results

The following refactoring transformations were tested using the semantic evaluator:

- Eliminate variable
- Rename function
- Rename variable

In addition to these refactorings, other transformations suitable for testing with the evaluator, with no or little extra effort, are:

- Extract function
- Generalize function definition*
- Move function
- Merge expressions
- Reorder function arguments*

The refactorings marked with an asterisk can only be tested with the current version of the evaluator if all function parameters are known. If not, the modified order of parameters causes a significantly different symbol table (as variable mappings are created in the order in which variables are encountered in the code) and thus a different semantic representation.

The following sections describe the three tested refactorings in detail.

5.4 Eliminate variable

We created a new module to test the *eliminate variable* refactoring for correctness. In the following example variable `Y` is eliminated and replaced by its assigned value.

```
1 f(X) ->  
2   Y = X + 1,  
3   g(Y).
```

Snippet 5.1: Eliminate variable - original function

```
1 f(X) ->  
2   g(X + 1).
```

Snippet 5.2: Eliminate variable - refactored function

To test this refactoring we analyzed all functions in a set of loaded modules for introduced variables. In this example, only Y is an introduced variable since X is an argument variable. Only introduced variables are candidates for elimination. For every introduced variable in the loaded modules we selected a random occurrence. Using the query language, we requested the corresponding cursor position in the source file. This cursor position is the parameter that is passed to the *eliminate variable* refactoring function.

Contrary to for example the *rename variable* refactoring, the number of possible *eliminate variable* refactorings per set of loaded modules is very limited. The number of candidate variables is finite and there is no need to generate a list of random names. We eliminated variables in a random order, until no further eliminations could be performed. Before and after every refactoring a semantic tree was generated using the symbolic evaluator. Before and after refactoring the semantic representation for all loaded functions should be the same. The implementation is as follows:

```

1 eliminate_var() ->
2   Files = ?QCCOMMONFUN:get_files_in_database(),
3   {_, RestorePoint} = ?Graph:backup(),
4   Result = perform_refactoring(generate_args(Files), Files),
5   ?Graph:restore(RestorePoint),
6   Result.
7
8 perform_refactoring([], _) ->
9   [];
10 perform_refactoring(Args, Files) ->
11   FilePath = ?QCCOMMON:get_file(Args),
12   CompResultBefore = compile:file(FilePath, [
13     strong_validation, return_errors]),
14
15   SemTreesBefore = generate_semtrees(Files),
16
17   Result = ?QCCOMMON:perform_transformation(reftr_elim_var,
18     Args),
19
20   TestResult =
21     case Result of
22     {result, _} ->
23       prop_compilation(FilePath, CompResultBefore) andalso
24       prop_semtree(SemTreesBefore, Files, OptionList);
25     _ ->
26       true
27     end,
28
29   [TestResult|perform_refactoring(generate_args(Files),
30     Files)].

```

Snippet 5.3: Erlang implementation of *eliminate variable* testing module

First, all files loaded in the database are requested. This list of files is used to generate the refactoring arguments. The implementation of `generate_args/1` is not shown here, but it returns a random pair of a file pointer and a cursor

position in that file. In line 3 a restore point for the semantic graph is created, which is used to restore the graph to its original state after refactoring.

The actual testing is done in the function `perform_refactoring/2`, which recursively calls itself with newly generated refactoring arguments, until `generate_args/1` returns an empty list. As with the implementation shown in snippet 3.1, the compilation result before refactoring is stored for later comparison. Also, a semantic tree of the original module is built on line 14. The rest of the implementation contains nothing that hasn't been discussed before. On line 27 the test results for every refactoring are simply enumerated in a list. In future implementations it is desirable to improve this data structure so that it holds more detailed info about the test results, like QuickCheck does. The function that performs the semantic tree comparison is the following one:

```

1 prop_smtree(SemTreesBefore, Files) ->
2   SemTreesAfter = generate_smtrees(Files),
3   lists:usort(SemTreesBefore) == lists:usort(SemTreesAfter).

```

Snippet 5.4: Erlang implementation of semantic tree comparison

The most important argument here is `SemTreesBefore`, which is a list of semantic trees, each of which belong to a single function. To compare the list of trees before and after refactoring, the `==` operator for syntactic equality is used. The lists are sorted to abstract from the order in which the semantic trees occur. Surprisingly, the simple equality operator serves our purpose quite well. During the whole project we have never felt the need to use a more elaborate equivalence relation to decide upon equivalence of semantic trees.

After calling the above function for snippet 5.1, the (non-canonized) semantic tree for the original function is:

```

{application,
  {g,
    [expr_choice,
      [{expr_choice,
        [{infix_expr,
          {'+', {variable, {v0}}, {constant, {integer, 1}}}]}}}]}

```

The semantic tree for the refactored function is:

```

{application,
  {g,
    [expr_choice,
      [{infix_expr,
        {'+', {variable, {v0}}, {constant, {integer, 1}}}]}}}

```

The two semantic representations are nearly the same, except for an extra `expr_choice` node in the first one. This is due to the implementation of the evaluation function that looks up the value for variable `Y`. This difference can be overcome by calling the evaluator with option `{canonical, true}`. Doing so, both the original and the refactored function will yield:

```
{application,
  {g,
   [{infix_expr,{'+'},{variable,{v0}},{constant,{integer,1}}]}}
```

The conclusion is that, for this simple case, the *eliminate variable* refactoring is correct, as far as semantic equivalence as defined in this paper is concerned.

5.5 Rename function / Rename variable

QuickCheck modules for the *rename function* and *rename variable* refactorings had already been written (see also chapter 3). To test these refactorings we only had to extend the corresponding modules with the QuickCheck property for semantic equivalence, of which the implementation is given in snippet 5.4. Skipping further implementation details, we discuss an example module with which both refactorings have been tested.

```
1 f(X) ->
2   inc(X).
3
4 inc(N) ->
5   N + 1.
```

Snippet 5.5: Rename function/variable - original module

In this example there are four candidates for refactoring: functions `f/1` and `inc/1`, and variables `X` and `N`. QuickCheck generates random names for these functions and variables and the module is refactored accordingly. After semantic evaluation using `{abstr_fun_names, true}` (since we want to abstract from function names), the representation for this module is:

```
[
{application,
  {function, [{variable, {v0}}]}},

{infix_expr,
  {'+'},
  {variable, {v0}},
  {constant, {integer, 1}}}
]
```

From this representation it becomes already clear that function and variable renaming, provided it is done correctly, does not change the semantic trees. As always, variables are referenced by their internal names and function names are replaced by the keyword `function`. However, there is no internal auto-numbering for functions as there is for variables, allowing incorrect *rename function* refactorings to go by undetected. In a future implementation this needs to be changed in order to improve the expressive strength of the semantic language.

A number of modules were tested for semantic equivalence before and after refactoring, and no errors were found.

Chapter 6

Discussion

6.1 Runtime errors

In the Erlang language it is quite common to see runtime errors while executing programs. For a large part these are type mismatch errors, caused by Erlang's dynamic typing system. When developing a static evaluator for Erlang one must be aware of the presence of these errors. In section 4.8 the case expression evaluation algorithm supports the presence of a non-matching type or value. This is an example of a runtime error that can be expected and therefore handled accordingly.

In addition to the “obvious” errors, there are many instances of runtime errors that are not so easily predicted. Even a simple addition can cause an integer overflow exception, making virtually every statement a potential source of errors. Consider the following example:

```
1 f(X) ->
2   X + 2 - 1.
3
4 g(X) ->
5   X + 1.
```

Snippet 6.1: Integer overflow example

Initially, one might conclude that functions $f/1$ and $g/1$ are semantically equivalent. However, this is not entirely true as $f/1$ causes integer overflow for $X = \text{MAXINT} - 1$, while $g/1$ does not. In this case, following standard terminology, $f/1$ would return *bottom* (denoted \perp). These more subtle aspects of semantics are not supported by the evaluator. The notion of bottom cases is completely left out.

6.2 Side effects

Many functions (at least in a functional language) are constructed in a way that they perform some actions on their input parameters and then return a value, without leaving any traces on the execution environment. These functions are said to be side effect free. Operations that do have side effects are the operations that modify the state of the program, for example by writing to a global variable, updating a database or modifying the file system. Deciding whether a program or function has side effects is not a trivial task; at the time of writing side effect analysis has only been implemented partially in RefactorErl. The query language includes a function `dirty/1` that, given a function node, returns a boolean indicating whether that function has side effects.

Development in the area of side effect analysis for RefactorErl is still ongoing and progress is being made. For every refactoring transformation the conditions state whether an expression should be side effect free in order to allow refactoring. The implementation, on the other hand, has not yet been perfected [18].

In the current implementation of the evaluator, side effects are completely ignored.

6.3 Explosion of possibilities

By implementation, branch statements and mathematical operations produce lists of values. These lists are kept and used for the rest of the evaluation process to produce new results, which are again lists. Starting with two symbols that both have two possible values, after a number of operations the number of possible values can become quite large. Suppose that two evaluated variables have m and n possible values respectively, then a mathematical operation like multiplication yields at most $m \cdot n$ values. Continuing to do so will result in exponential growth of the list of possibilities. This will cause a significant performance penalty.

In [15], Lindahl and Sagonas experience the same problem and they solve it by collapsing a list of possibilities into a more general representation, such as `integer()`, if the number of items in the list exceeds a certain threshold. Another solution proposed by them is to have ranges instead of separate values, for example [0...21].

Chapter 7

Conclusions and recommendations

7.1 Introduction

In this thesis we have presented a partial solution to the problem “how can correctness of refactoring transformations be tested?”. Instead of defining test cases for each refactoring transformation we have developed a platform which is an abstraction from the Erlang syntax and therefore forms a basis for further testing. In the following sections we will evaluate the solution and give a number of topics for future research.

7.2 Personal thoughts

Doing this research has provided an excellent insight in the ways of working of an academic research team. In addition to a number of full-time RefactorErl developers, a group of master students worked on the project as well in the context of a course they were doing. This connection between the research-focused and educational part of a university is a strong feature of the way the project is organized. The weekly team meetings provided a lot of context for the specific task that we worked on and allowed us to see the bigger picture.

As with many master theses, it took quite some time to get comfortable with all the tools that played a role (Erlang, RefactorErl, the query language, Emacs, QuickCheck). Our colleagues supported us very well by answering all our questions, but still the complexity of the material caused a “startup” time of several weeks.

Ely Deckers and I were more or less free to come up with and develop our own ideas, which is comfortable, but I feel that a little more guidance could have resulted in a solution that better integrated with the rest of the team’s efforts.

7.3 Strong points

We consider the following properties to be strong points of the proposed solution:

1. Instead of providing case-by-case solutions for testing every refactoring transformation individually, we have created a platform that can be used for testing. Existing modules that use QuickCheck and the query language in order to test a specific refactoring can now be extended with an extra property: semantic equivalence.
2. A list of options that influences the evaluator’s behavior allows for usage with many refactorings. For example, in the general case it might be desirable to include function names in the semantic tree. However for the *rename function* refactoring one needs to abstract from specific names. Setting the corresponding flag allows a tester to do this.
3. The evaluator has been used successfully in a number of real-life tests. This proves that the evaluator itself works and, more importantly, has strengthened the confidence in the correctness of the refactorings under test.
4. The evaluator could contribute to the thoroughness of data flow analysis [19]. Data flow analysis is a separate branch of research within the RefactorErl project that tries to predict which parts of a program are affected when another part is being changed. The symbolic evaluation and pattern matching technique created for the evaluator could benefit this analysis.

7.4 Weak points and future research

Naturally, there are a number of weak points as well. We consider the following points eligible for improvement:

1. The evaluator in its current state only handles a very small subset of the Erlang language. This makes it relatively useless for most real-life applications. Experience has taught us that if there are errors in a refactoring transformation, these occur in the easily overlooked, more “exotic” language constructs. An example has already been given in section 3.3, where we showed that a deprecated syntax for function calls wasn’t properly refactored.
2. The evaluation of function calls isn’t powerful enough. The function clause matching algorithm is too strict, in the sense that it cannot handle variable uncertainty for function arguments. Basically the function clause matching problem is equivalent to pattern matching in case expressions, but it hasn’t been implemented as such. In the future it is desirable to unify these algorithms to provide a generic approach for pattern matching.

3. The abstraction in function names weakens the expressive power of the evaluator, and can cause two trees to be syntactically equivalent while the corresponding functions are semantically different. Instead of simply using the keyword `function`, it is desirable to indicate *which* function is called, for example by mentioning its node-ID in the semantic graph.
4. Side effects analysis, support for message passing and concurrency and other advanced topics are not implemented. If one were to further develop the evaluator into an industrial-strength application these would be necessary features. However these are well beyond the scope of this thesis.

Bibliography

- [1] Joe Armstrong. A History of Erlang. 2007. http://www.cs.chalmers.se/Cs/Grundutb/Kurser/ppxt/HT2007/general/languages/armstrong-erlang_history.pdf.
- [2] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [3] Eclipse. <http://www.eclipse.org/>.
- [4] Microsoft Visual Studio. <http://microsoft.com/VisualStudio/>.
- [5] HaRe - The Haskell refactorer. <http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html>.
- [6] A list of publications about RefactorErl. <http://plc.inf.elte.hu/erlang/>.
- [7] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. Refactoring with Wrangler, updated - Data and process refactorings, and integration with Eclipse. *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, 2008.
- [8] László Lövei. E-mail conversation, 18-03-2010.
- [9] László Lövei, Zoltán Horváth, Zoltán Csörnyei, Tamás Kozsik, Roland Király, Róbert Kitlei, István Bozó, Csaba Hoch, Melinda Tóth, Dániel Horpácsi, Dániel Drienyovszky, and Kornél Horváth. Implementation of Erlang Refactoring. *Unpublished*, 2008.
- [10] István Bozó, Gergo Érdi, Lilla Hajós, Zoltán Horváth, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, Tamás Nagy, and Melinda Tóth. Complexity metrics and simple semantic queries for Erlang. *Unpublished*, 2009.
- [11] István Bozó, Dániel Drienyovszky, Dániel Horpácsi, Kornél Horváth, Zoltán Horváth, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, and Melinda Tóth. Progress report on Erlang Refactoring. *Unpublished*, 2009.
- [12] Ely Deckers. Verifying properties of transformations in RefactorErl using QuickCheck. Master's thesis, Radboud University Nijmegen, 2010.

- [13] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000.
- [14] Quickcheck. <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.
- [15] Tobias Lindahl and Konstantinos Sagonas. TypEr: a type annotator of Erlang code. *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, 2005.
- [16] Dialyzer. <http://www.erlang.org/doc/man/dialyzer.html>.
- [17] Jonas Barklund and Robert Virding. Erlang 4.7.3 Reference Manual, Draft 0.7. 1999. http://www.erlang.org/download/erl_spec47.ps.gz.
- [18] László Lövei. E-mail conversation, 05-05-2010.
- [19] Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik. Impact analysis of Erlang programs using behaviour dependency graphs. *Unpublished*, 2009.