Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Refactorings towards clean functional code[1]

TDK thesis

Supervisor:
Melinda Tóth, István Bozó
Associate professor, Assistant professor

Author:
Boldizsár Poór
Computer Science BSc
2nd year

Budapest, 2020

[1]The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Abstract

The programming style has an impact on the readability and comprehensibility of the source code, and it may have an effect on the run-time performance. This statement also holds for functional languages when the functional style is mixed with imperative design. In this paper, I present a couple of methods, that can refactor imperative style Erlang source-code into a more functionally styled structure. This can be done by transforming unnecessary calls to `length`, `hd` and `tl` into pattern matching or by lifting particular nested expressions. The results of various measurements indicate that these refactorings can not only shorten the length of the source code, but can also relevantly affect the execution time. Accordingly, these methods will be implemented and should be extended with refactorings of similar patterns from which a total source code cleanup methodology could be worked-out.

Contents

Chapter 1

Introduction

Developing a program in functional programming language is not always easy. It is especially not when someone has origins from the imperative world. Such people often find it challenging to acquire the functional mentality. It is perfectly reasonable since they are used to the other perspective.

Because of this, one can find many style mismatches when reading the code of a newcomer and may want to transform the code into a more functional one. Hence, I am going to define a couple of refactorings that can help people in such situations.

1.1 Background

Before we begin, understanding a couple of jargon phrases will be essential in understanding the thesis. Therefore we will go through the background required before all.

1.1.1 What is refactoring?

Code refactoring is the process of restructuring existing source code without changing the function of it. [1]

Its heart is a series of small behavior preserving transformations. Each transformation does little, but a sequence of these transformations can produce a significant restructuring.

The fundamental purpose of code refactoring is to make the code more readable, maintainable and also to improve extensibility.

1.1.2 What is Erlang?

Erlang[2] is a general-purpose programming language with built-in support for concurrency, distribution and fault tolerance developed by Joe Armstrong, Robert Virding, and Mike Williams in 1986 originally as a proprietary software within Ericsson, but was released as free and open-source in 1998.[3]

The term Erlang is used interchangeably with Erlang/OTP or Open Telecom Platform which is a collection of useful middleware, libraries and tools written in the Erlang programming language. It is an integral part of the open-source distribution of Erlang.

BEAM (Bogumil's/Björn's Abstract Machine) is the virtual machine at the core of Erlang that is included in Erlang/OTP. [4][5]

Nowadays, other programming languages such as Elixir or LFE run on the BEAM virtual machine.

1.1.3 What is the RefactorErl project?

RefactorErl is an open-source static source code analyser and transformer tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary.[6]

The main focus of the project is to support the daily code comprehension tasks of Erlang developers.

The tool has a custom Erlang source code analyser and transformer that is capable of handling real-world code. Features include support for analysing macro constructs, storage and fast retrieval of analysis results, and source code layout and comment preservation during transformations.

1.2 My Goal

Recently, I have got to know some popular IDEs such as Visual Studio, CLion and PyCharm, which are all amazing. They have nice syntax highlight, great debugging tools and also they have built-in refactoring tools, that not only will help you rename a variable or execute simple refactorings, but will also suggest you, how you may write cleaner, more readable and effective code. Also, these suggestions are shown while writing your program, that is incredible.

In contrast to this, popular functional programming languages like F#, Haskell, Scala or Erlang have much weaker tool support for the programmers. We may find some nice plug-in for a good IDE or use Emacs that does have amazing support, but for newcomers to the functional world, they are hard to get used to.

My goal is taking part in creating better and more supportive environment for programmers of the functional world and thus make the future generation find the it more attractive. Thus the friendly community of the functional languages could get bigger and more people would discover the beauty of composing a functional program.

1.3 Results

In this paper, I present three transformation methods.

The first one is transforming a guard to pattern which eliminates redundant calls to the `length` function and uses pattern match, when possible. When this refactoring is applied, the execution time of the program will be improved without any semantic change in it.

The second is the branch lifting, for which I defined the steps and conditions of application for all possible cases of containment. The application of this refactoring can improve readability.

Thirdly, I present the transformation, that can eliminate the calls to `hd` and `tl`.

I present where the refactorings could be applied and give a method for looking for instances for two of them, and implemented a query, that can find the last one. Furthermore, I measure how each transformation affects different software complexity metrics, and create visualisations from the results.

Moreover, I implement these refactorings in the RefactorErl framework.

Lastly, I presented these ideas in the 3in Carpathian Basin Conference on Software and Security, 2019.

1.4 Related work

Refactoring is generally a popular topic in studies. We can sate the same concerning functional languages. In this section, I am going to present some tools developed for refactoring functional code and their functionality in a nutshell.

HaRe

HaRe [7] is a refactoring tool for Haskell programs developed at the University of Kent.

It supports a great set of refactorings over Haskell. To specify some it supports structural refactorings such as removing an unused variable or renaming a function. It also supports module-aware refactorings taking care of imports and exports or moving from one module to another. Data-oriented refactorings are also present in the system.

Unfortunately, HaRe will only work for projects using GHC 7.10.2 for compilation.

Wrangler

Wrangler [8] is an interactive refactoring tool for Erlang programs.

Wrangler supports quite a lot refactorings such as renaming variable / module / function, generalising function definition, moving function definition to another module and many more.

Wrangler is integrated in Emacs and Eclipse.

Tidier

Tidier [9] is another refactoring tool for Erlang programs. The main design characteristic of Tidier is that it is a fully automatic system, unlike RefactorErl or Wrangler.

It automatically modernizes guards and calls to old-fashioned functions, transforms a couple of function from the `lists` module into a list comprehension, it reduces redundancy in many places and has many more little transformation. It has some very specific cases to transform as well.

Overall, I like what it does. The unfortunate thing about this project is that it ran on a website and for two years it could not be accessed.

Chapter 2

An example with Ackermann

In the first years of the university we are taught how effective programs are designed and the background. Nonetheless, one can see a couple of patterns, that occur quite some times and are rather annoying to see, since they are highly inefficient. To demonstrate what I mean we will go through the following example:

2.1 Ackermann function on lists

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

This is the two-argument Ackermann–Péter function[10]. Currently, we want to implement this function in a slightly different way: n and m is going to be a length of a list, so for instance $m = 0$ will be $(length(M) =)m = 0$ namely M is an empty list. The `+ 1` will mean inserting an element to the list and thus increasing its length by one. The `- 1` means to remove one element form the list so its length will be smaller by one.

0In the examples below we will work with a atoms as the only member of the lists.

With the given specifications our first implementation can bee seen on Figure 2.1

At this point, we could stop as we have a proper function that does what it should, but we want to have a better implementation. This function is massively recursive, so every decision we make, every function we call and even the tiniest piece of change can significantly change the execution time of the code.

In Erlang, the lists we work with are linked lists. This is good in functional languages because you can prepend to an immutable singly linked list in $\mathcal{O}(1)$ time. This can help us to recursively build

```
ack( M, N ) ->
    if
        length(M) == 0 -> [a|N];
        length(N) == 0 -> ack(tl(M), [a]);
        true           -> ack(tl(M), ack(M, tl(N)))
    end.
```

Figure 2.1: First Ackermann implementation

up n-element lists in $\mathcal{O}(n)$ time. On the other hand, some methods will have $\mathcal{O}(n)$ complexity instead of $\mathcal{O}(1)$, such as indexing and calculating the length.[11]

So now we see, that we should consider not using the `length` function when it is possible, but what can we do then?

Erlang gives us a marvellous tool, pattern matching that could be used instead.

Pattern matching

Pattern matching is the fundamental construct in functional languages. A simple pattern matching expression looks so:

```
pattern = expression.
```

This will evaluate the expression, and will try to match the result on the given pattern. It will result in an error if the pattern match does not succeed.

Generally, pattern matching succeeds in the following cases:

- The pattern is an unbound variable. When the pattern match succeeds, the variable will be bound to the value of the expression it was matched with.
- The pattern is a constant or variable bound to a value. In this case, the pattern match will succeed when the value and the type of the expression and the value in the pattern equals.
- The pattern is a structure (e.g. list or tuple) which may contain other expressions to pattern match on. It will succeed when the structure of them are the same and also the patterns inside the structures all succeed.

Case expressions

The case expression uses pattern matching, and has the following syntax:

```
case Expression of
    Pattern1 [when Guard1] -> Expression_sequence1;
    Pattern2 [when Guard2] -> Expression_sequence2;
    ...
    PatternN [when GuardN] -> Expression_sequenceN
end
```

Figure 2.2: Syntax of a case expression

The brackets indicate that the guards are optional.

The expression in the head of the `case` is evaluated, and we will try to match on the patterns in order. When a matching pattern is found where the guard is also evaluated to `true`, the body (an expression sequence) of the branch will be the result of the `case` expression. [12]

With a case expression, we can easily construct a new implementation where we eliminate the usage of length by using case expressions. (Figure 2.3)

```
ack( M, N ) ->
   case M of
       [] -> [a | N];
       M ->
           case N of
               [] -> ack( tl(T), [a] );
               N  -> ack( tl(T), ack(M, tl(N)) )
           end
   end.
```

Figure 2.3: Ackermann in nested case

As we will see on Table 2.1, this particular transformation was an immense performance-enhancement and further optimization is rather irrelevant.

Currently, our problem is low-level of readability. It is a rather hard task to understand what is the function of this code and we want to make high-quality code so we will try to restructure it a little.

After a little bit of research, we can find great resources about Erlang programming conventions. These conventions are important if we want to write elegant code. One of these conventions is "Don't write deeply nested code", that we now ignored in the previous transformation.[13]

By eliminating nesting, we could improve readability. To achieve this, we are going to use the tuple of M and N instead of using them as a single value, thus we can decrease nesting level by one.

This version seems much friendlier, and this would be totally okay, but our goal is the absolute perfection, so we will try to develop it even more.

Pattern match can not only be used in case expressions, but we can also use it in many places.

```
ack( M, N ) ->
    case {M, N} of
        {[], _} -> [a | N];
        {_, []} -> ack(tl(M), [a]);
        _ -> ack(tl(M), ack(M, tl(N)))
    end.
```

Figure 2.4: Ackermann in tuple pattern

We can write very similar pattern matching in function heads.

There are a few differences, but having more than one expression we can pattern match on is relevant to us right now. One may write a function with several patterns where the first matching on all patterns will be the result.

Knowing this, we can one more time, decrease the nesting level of the implementation and get to see a much shorter one.

```
ack( [], N ) -> [a | N];
ack( M, [] ) -> ack(tl(M), [a]);
ack( M,  N ) -> ack(tl(M), ack(M, tl(N))).
```

Figure 2.5: Ackermann with function pattern

As we glance through the new code, it almost feels like we reached our final goal, but we may notice one last discomforting piece of code.

As it was mentioned before, the ultimate weapon we are using, pattern-matching can also match on a little more compound expression. The head of a list is the first element in that list, and the tail of it is every element in that list, except from the first. Matching a list with at least one element (i.e. its length is larger than one) on a head-tail pattern will succeed, but an empty list will never match on this pattern.

```
ack([], N) -> [a|N];
ack([_|M], []) -> ack( M, [a] );
ack([_|MT]=M, [_|N]) -> ack(MT, ack(M, N)).
```

Figure 2.6: Ackermann without hd and tl

Ultimately, our last Ackermann function eliminates both length and tl calls, looks readable without having to spend much time understanding the implementation and is relatively short.

2.2 Measuring execution time

After the transformation, I measured the execution times for each definition with arguments `[a, a, a]`, `[a, a, a, a, a, a, a]`. The average of 1000 calls of each implementation can be seen on Table 2.1.

Function version	**Time**
If with length calls	172499.438 ms
Nested Case	9325.02 ms
Case with tuple	9484.958 ms
Function pattern	9326.604 ms
Function pattern without `hd` and `tl` calls	6619.282 ms

Table 2.1: Execution time for each implementation (average of 1000 calls)

The results – as it was mentioned before – indicate that the most significant performance enhancement is due to the elimination of `length` calls.

Unfortunately, this optimisation made the code much less readable. The solution for that problem was un-nesting, where we could see two types of that.

One was `case` in `case`, where we reduced nesting by using tuples for patterns. The other was `case` in `function`, where we moved the patterns to the function head and thus were able to create a nesting-free source-code.

In the following chapters, we will examine each transformation, as a possible refactoring, and when possible, will work out the methodology and conditions of applicability for it.

Chapter 3

Transforming guard to pattern

People who are used to programming in imperative languages often use the function `length` to decide whether a list is empty or not. Semantically, there is no problem with this.

If we want to define the base case of a recursive function when we have got an empty list, this strategy will work perfectly fine. Nevertheless, considering the wasted computing power, it does bother people developing their code in functional style.

Because of this, frequently seen imperative pattern had I made up my mind to work out a new refactoring methodology, eliminating the redundant calls to the function `length`.

3.1 Transformation steps and conditions

To define this refactoring, we will analyze a couple of expressions for transformability. When one expression can be transformed, we will look into further details such as the steps of the transformation and conditions of applicability.

3.1.1 The base of the refactoring

The whole refactoring originates from a very simple idea.

When you see someone using the above mentioned pattern – checking equality for the `length` of a list – but with 0, you can easily transform that into a `case` expression with the case head of the list and a branch with the pattern of an empty list.

Transformable expressions

After brain-storming about this refactoring, the idea got a little bit extended. Not only the number zero could be transformed, but any constant number written directly in the source-code can be eliminated next to a `length` call by using wildcards in the pattern.

Wildcard or otherwise known as joker pattern is a kind of placeholder represented by a single character ('_' in Erlang) which matches on any value. This pattern is usually used when the value that would be bound in the pattern would not need to be used. Instead the wildcard will match on anything and will not bind any value to itself.

When a pattern is a list containing n wildcards, it will match strictly on lists of the length n. Thus checking equality of the `length` of a list with any constant integer can be transformed into pattern matching.

Beyond that, we are given the possibility in almost all functional languages to write patterns on the head and tail of a list (the head is the first element of a list, and the tail is the rest). What is more, we can write unlimited heads in a pattern, taking that number of elements form the front of a list, and leaving the rest in the tail.

Having thought about this, we can not only transform the operator `==`, but using head and tail patterns we can also refactor the checks if the length of a list is greater or equal to a constant integer by taking n wildcards as the head and one last as the tail of a list.

There is one thing need to be mentioned. We can not transform an expression checking if the length of a list is greater or equals to zero with the above specified procedure since it can not be done with our above specified tools. This kind of code can rarely be found since it's meaning is basically is that thing a list? We can use the `is_list` function in the guard since the semantic meaning of the code will not be changed thus.

Currently, the `==` and the `<=` operators can be transformed (when the left-side value is the length), but we would like to expand the conversion for `<` as well.

$$\forall n, m \in \mathbb{N} : n \leq m \iff n < m + 1$$

Considering the above equation we can transform like-wisely an expression with simple "is greater" checks.

One last transformable expression to be defied before moving forward is the `true` guard. If we want to transform a `true` guard – that behaves like an otherwise branch – than having an expression that will always match on any pattern will be its equivalent in a case expression. The wildcard pattern is perfect for this job so the transformation of a `true` branch will be a wildcard pattern.

Non-transformable expressions

So far, we have only been discussing transformable expressions, but there are of course non-transformable ones as well.

Using a very simple example the expression `N < 3` can not be transformed into pattern matching. When we find such guards in an `if` that needs to be transformed what should be done?

We can approach this problem in two different ways.

The first option when finding a non-transformable guard is cutting the `if` into two by the found expression and taking the second half of the branches to the body of a new `true` guard in the first half `if`. Then, we can transform the first – now outer `if` – into a case expression since all guards before the specified one were transformable.

Then we may try to refactor the inner if as well. We will have to look for the first transformable guard in there. If the first one is a `true` guard that is also the last branch of the `if` then we leave it that way. Otherwise we will again cut the `if` expression at the found guard and continue the transformation.

Generally, we cut the if every time a transformable and a non-transformable guard meets.

After this, we may get a deeply nested code, but since we have defined the refactoring for lifting `if in case` in Section 4.2 and `case in if` at Section 4.4 we will be able to lift this expression, thus flatten the nesting. Than we are done.

The other approach will have the same result. When we find some non-transformable guard, we will move that to a guard of the case branch. Then we need a pattern, that will match on anything. And one more time, we can use the wildcard pattern since that is perfect for this purpose too.

Transforming `'=:='`

The operator `=:=` is basically analogous to pattern matching, so it is very easy to transform. Simply move one of the sides (for readability reasons, it should be a variable) to the case head, and the other to the pattern. Thus we have one more transformable pattern.

This will work perfectly with anything but what about the operator `==` ?

The interesting case of `'=='`

In Erlang, if someone wants to test precise equality namely equality of type and value, then `=:=` have to be used.

When someone wants to compare two numbers, whether their value is the same, or not you can use the `==` which will convert the term with the lesser precision into the type of the other term.

A float is more precise than an integer until all significant figures of the float are to the left of the decimal point. The conversion strategy is changed depending on the size of the float because otherwise comparison of large floats and integers would lose their transitivity.[14]

For this reason you should use =:= and =/= when it is your intent to compare same terms and use == and /= only if it is your intent to compare numbers.

Due to these, the transformation of `==` (or `/=`) will only be applied if no integer or float value is found in the term, since we would need to create patterns for both float and integer typed term for every number.

Theoretically, if we wanted to transform such expression, we would need to create a pattern for each possible variation of the types with the same body as in the original expression. Thus finding n numbers in an expression would result in 2^n branches with the same body. This would be highly unreadable so this kind of expression will not be transformed.

After collecting these ideas, a refactoring showing the above defined ideas can be seen on Table 3.1

```
base_from(List, N) ->
   if
       length(List) == 0 -> expr1;
       length(List) > 3 -> expr2;
       List =:= [atom1, 1] -> expr3;
       length(List) >= 1 -> expr4;
       N > 2 -> expr5;
       true -> expr6
   end.
```

```
base_to1(List, N) ->
   case List of
       [] -> expr1;
       [_,_,_,_|_] -> expr2;
       [atom1, 1] -> expr3;
       [_|_] -> expr4;
       _ ->
           if
               N > 2 -> expr5;
               true -> expr6
           end
   end.
```

```
base_to2(List, N) ->
   case List of
       [] -> expr1;
       [_,_,_,_|_] -> expr2;
       [atom1, 1] -> expr3;
       [_|_] -> expr4;
       _ when N > 2 -> expr5;
       _ -> expr6
   end.
```

Table 3.1: The base of the transformation

The function `base_from` contains an `if` that has a guard as an example for each above written transformation.

`base_to1` is the temporary state of the firstly defined non-transformable transforming strategy, and `base_to2` is the final result for both.

Guards in `case` expressions

As we will see in Section **??**, a pattern one can see rather frequently is calling `length` in the guard of a `case` expression. Previously, I only described the refactoring for guards in an `if` expression. Now, I am shortly going to describe how to eliminate guards in a `case` expression.

Suppose, there is a transformable expression in the guard of a branch. We will cut the `case` expression after this expression. Then we build an `if` expression in the body of the specified branch. We move the guard to this `if` expression and create a `true` branch after. The body of the original branch will move next to the guard and the second half of the cut will move to the body of the `true` branch.

Finally, we can apply the refactoring to the `if` expression that contains the transformable guard, which result in a `case in case`. By applying the branch lifting refactoring, we get a non-nested and transformed `case` expression and we are done.

3.1.2 Negation

So as we have seen, '==', '=:=', '>' and '>=' can be transformed, but can we go any further? By negating the upper expressions, we could achieve transforming any comparison operators into much faster pattern matching.

Suppose, we have an `if` expression with a guard that contains an expression like `=/=` or `<`. Generally, an expression that could be transformed if we used its negation.

In such cases, we will negate the specified guard, transform it and create a new wildcard branch after that. After this, we place the negated expression's body in the next – wildcard – pattern's body and insert the rest of branches – that we may want to transform – to the body of the negated guard.

If we match on the negated expression, it means that it would originally evaluate to `false` and thus the program would continue the evaluation of the guards and this is exactly, what the program will do.

If we did not match on the expression we will match on the wildcard and the result will be its body which is the original guard's body.

Thus we specified the refactoring for any guard, that has a transformable negation.

```
negation(List) ->
    if
        length(List) =/= 2 ->
            expr1;
        List =:= [a, True] ->
            expr2;
        true ->
            expr3
    end;
```

```
negation(List) ->
    case List of
        [_, _] ->
            case List of
                [a, True] -> expr2;
                _ -> expr3
            end;
        _  ->
            expr1
    end.
```

Table 3.2: Transforming a negated expression

The unfortunate drawback of this transformation is nesting and one more thing. The expression `Var =/= [a, b]` is significantly more readable than its transformed form.

For this reason, the refactoring with negation is only recommended to be applied, when a `length` call can be eliminated by this.

There is also one more thing, that need to be thought over. What if the guard that we would normally negate to transform is the last in the `if`? Than we can not transform the guard since we have nothing to move to the body of negated-transformed branch, so after transforming everything that we can we leave this guard untouched.

3.1.3 Difference between eager and lazy operators

In Erlang, you can chose from using eager operators (`or`, `and`) or you can use the lazy ones (`orelse`, `andalso`). The eager operators will evaluate both sides, while the lazy ones will evaluate the first and the second expression will only be evaluated if necessary.[15]

Based on this, if the left side of the expression has side-effect, the program will behave differently using the lazy and the eager operators.

What currently interests us is how will we need to transform these expressions not to change any semantic behaviour of the program.

Fortunately, we do not have to deal with this problem because Erlang has an interesting policy on side-effects in guards.

3.1.4 Side-effects in guards

In Erlang, guards behave differently than standard expressions [16]. Only a subset of Build-In Functions (BIFs) is allowed to use when writing a guard. These are the ones that are side-effect free such as `is_list` or `length` [17].

The more subtle difference is handling of errors. In normal expressions, an error crashes the process if not caught in some way. In guards, however, the error will not ruin the process, the only result is that the guard fails and we continue the evaluation branches.

This means the following expressions are not equivalent.

```
side_effect(List) ->
    if
        length(List) == 0
              -> expr1;
        false -> expr2
    end.
```

```
side_effect(List) ->
    case length(List) == 0 of
        true  -> expr1;
        false -> expr2
    end.
```

Table 3.3: The comparison of `if` and `case`

If we call the function implemented on the left side with something that is not a list, it will return with `expr2`. In contrast, doing similarly with the function implemented on the right side, an error will kill our process.

3.1.5 `or` and `orelse`

As we have seen, guards can not have side-effects and thus we will not have to deal with difficulties it would cause.

Having seen that all comparison operators can be transformed into pattern-matching now we want to think a little bit about `or` and `and`, but firstly we will examine the former.

In transforming an `or` expressions the transformability of the sides plays a major role. Thus the methods will be divided into the following cases.

The first case is when we have two transformable expressions in both sides of the `or`. In such cases, we will create two branches with the body of the original branch, and move the expressions form the `or` to the guards. Then both sides can be transformed and the refactoring can be executed.

The second case is when we have two expressions in the `or` where one can and one can not be transformed. We will begin likewise, as we did before. We create a new branch for both sides with the original body but after this we will only transform the one, that can be transformed.

The third and last case is having two non-transformable expressions in the `or`. This is the easiest case, since what we have is a simple non-transformable expression that happen to contain an `or` expression. Therefore we have the case described in Section 3.1.1 and no new method needs to be worked out.

```
or_from(List, N) ->
   if
       (length(List) == 0) or (length(List) == 1) -> expr1;
       (length(List) >= 1) or (N > 2) -> expr2;
       (N > 2) or (N < 10) -> expr3;
       true -> expr4
   end.
```

```
or_to(List, N) ->
   case List of
       [] -> expr1;
       [_] -> expr1;
       [_|_] -> expr2;
       _ when N > 2 -> expr2;
       _ when(N > 2) or (N < 10) -> expr3;
       _ -> expr4
   end.
```

Table 3.4: Transforming each possible case of an `or` expression

3.1.6 `and`, `andalso`

Maybe the most interesting and complicated case is dealing with an `and` expression. First of all, as we had with `or`, we have three different cases depending on the transformability of the sides.

Let us now begin with the easy one, having two non transformable expressions. We will only have to move the whole expressions into a case branch wildcard pattern – just like with `or` – and we are done basically.

The little more complicated case is the mixture of transformability in the sides. In the beginning the idea was based on nesting. It was to create a `case` expression from the transformable side. If that has two branches – one is the transformation result and the other is a wildcard – than we move the rest of the `if` expression into the first branches body with the other side of the `and` as well and the same for the second branch but without the other side of the `and`.

In this case, when we match on the first pattern we have the possibility to have the result of the `and` expression and everything else, but when not we can only have the results from the rest of the expression. This kind of transformation works fine with the disadvantage of decreasing the level of code readability and duplicating parts of the source code.

The better and little later worked out method is creating tuples in the head of the `case` expression and use pattern matching on both of the expressions in a tuple. Using this, the code will be much more elegant and also it may be easier to transform into a function with patterns later on.

The firstly described transformation of `and_from` function can be seen in `and_to_1` and the second in `and_to_2` on Table 3.5

```
and_from(List1, List2, N) ->
   if
       (length(List1) == 0) and (length(List2) == 1) -> expr1;
       (length(List2) >= 1) and (N > 2) -> expr2;
       (N > 2) and (N < 10) -> expr3;
       true -> expr4
   end.
```

```
       and_to_1(List1, List2, N) ->
          case List1 of
              [] ->
                  case List2 of
                      [_] -> expr1;
                      [_|_] when N > 2 -> expr2;
                      _ when(N > 2) and (N < 10) -> expr3;
                      _ -> expr4
                  end;
              _ ->
                  case List2 of
                      [_|_] when N > 2 -> expr2;
                      _ when(N > 2) and (N < 10) -> expr3;
                      _ -> expr4
                  end
          end.
```

```
          and_to_2(List1, List2, N) ->
             case {List1, List2} of
                 {[], [_]} ->  expr1;
                 {_, [_|_]} when N > 2 -> expr2;
                 _ when (N > 2) and (N < 10) -> expr3;
                 _ -> expr4
             end.
```

Table 3.5: Transforming each possible case of an `and` expression

3.2 Summary of the conditions of applicability

The great thing about this refactoring is that it can almost every time be applied but of course there are some edge-cases when the application of the transformation would result in changing the meaning of the program. Having such case, we will deny applying the transformation.

To begin with, the `==` with a variable on the left and any number in the right side is the case when the expression will be marked as non-transformable.

Transforming a `length(L) >= 0` will result in either an `is_list(L)` or the transformed form of `length(L) > 0 or length(L) == 0` depending on the user's preferences. Both will preserve the meaning when the expression is located in a guard thanks to no error policy described in Section 3.1.4.

The elimination of `length` were written with examples where we call this function with values bound to variables. The question may arise, what if it was not a variable but some calls to functions.

Since no functions can be called from the guards except from a subset of BIFs we need to take a look at the allowed ones [17]. After looking through all of them, they can only return two things, either a Boolean value or a number as the size of some structure. Since asking for the length of a number or a Boolean can only result in an error, the refactoring can not be applied when a function is called inside the `length`.

When we transformed the expression with negation we needed to have an expression to move to the new body of the negated expression. Because of this, we need at least one branch after the one to be negated. When no branch is found after, the branch will be marked as non-transformable.

Chapter 4

Lifting nested `case` and `if` expressions

Nested code is code containing `case`/`if`/`receive` statement within another `case`/`if`/`receive` statement. According to Erlang programming guidelines, one should avoid nesting since it can hinder readability, maintainability, debugging and writing unit tests. [13] [18]

There are special cases when nesting can be substituted while preserving the semantical meaning of the code. Still, refactoring tools for functional programs do not support such refactoring.

In the subsequent chapter, we describe the methodology for lifting an `if` or a `case` expression from some particular code environment.

Representation of the program

Erlang will be used to represent the programs, and to write the refactoring algorithms as well. It was chosen, because, with atoms, one can intuitively recognise the represented programs and thanks to pattern matching, the algorithms are clear and concise.

To represent a program, let us be given guards, patterns, and expressions.

Let us represent a `pair` as a triple constructed from an `tuple` atom and two expressions, potentially guards.

Let us represent an `if` expression as a pair of an `'if'` atom, and a list of `if` branches where an `if` branch is a pair containing a guard and the corresponding body sequence.

Let us represent a `case` expression as a triplet. Its first element is a `'case'` atom, its second element is the head of the `case` expression, and its last element is a list of `case` branches where

a `case` branch is a triplet constructed from a pattern, a guard and the corresponding body sequence.

Lastly, let us represent a function clause as a pair. Its first element is a pair constructed from a `'fun'` atom and the name of the function. The function clause's second element is a list of function branches where a function branch is a triplet constructed from a list of patterns, a guard and the corresponding body sequence.

The six cases

There are six type of nestings, that we will transform.

In the followings, independent un-nesting algorithms are given for each type of nesting. In each case, the `reduce_nesting` function shall be called to transform an expression containing nested `if` or `case` statement. Lastly, we combine all of them in one function using the defined helper functions.

4.1 `if` in `if`

Listing 4.1: An algorithm to lift an `if` expression from an other

```
1  reduce_nesting({'if',
2    [IfClause = {_, [{'if', _}]}|OuterIfClauses]}) ->
3      {'if', if_in_if(IfClause) ++ OuterIfClauses};
4  reduce_nesting({'if', [IfClause|OuterIfClauses]}) ->
5      {'if', NewIfClauses} = reduce_nesting({'if', OuterIfClauses}),
6      {'if', [IfClause|NewIfClauses]}.
7
8  if_in_if({OuterGuard, [{'if', InnerClauses}]}) ->
9    [{both_true(OuterGuard, InnerGuard), InnerBody}
10     || {InnerGuard, InnerBody} <- InnerClauses].
```

The algorithm on Listings 4.1 - namely the `if_in_if` part of the refactoring - finds the first lift-able nested `if` expression, then it connects the outer guard with all the inner guards using the `both_true` function.

The `both_true` function connects two guard with an `and` operator, but can be improved by only using the `and` operator if non of the guards are a `true` condition.

Note that there are requirements of applicability for preserving the semantical meaning of the code, and the code quality.

Namely, one such precondition is the inner `if`, as a single expression shall be the body of an other `if` clause. It is therefore justified not to create unreasonable code duplication. This is not

checked in the algorithm, but it will only transform such clauses.

Another requirement is that the inner `if` statement must always have at least one guard evaluating to `true`. It is needed because if all of the inner guards evaluate to `false` then Erlang throws an `exception error: no true branch found when evaluating an if expression`, and recreating it is hard, unpleasant and unreadable. This precondition is not checked in the algorithm, but could be in the `if_in_if` function.

On Table 4.1 we demonstrate an example for lifting an `if` expression from an other.

```
if_in_if_from(X, Y, Z) ->
    if
        X < Y ->
            if
                X + 5 < Z ->
                    expr1;
                true ->
                    expr2
            end;
        X > Y ->
            expr3;
        true ->
            expr4
    end.
```

```
if_in_if_to(X, Y, Z) ->
    if
        (X < Y) and (X + 5 < Z) ->
            expr1;
        X < Y ->
            expr2;
        X > Y ->
            expr3;
        true ->
            expr4
    end.
```

Table 4.1: An example for lifting an if expression nested in an other

4.2 `if` in `case`

Listing 4.2: An algorithm to lift an `if` expression from a `case` expression

```
1  reduce_nesting({'case', Head,
2    [CaseClause = {_, _, [{'if', _}]}|OuterCaseClauses]}) ->
3      {'case', Head, if_in_case(CaseClause) ++ OuterCaseClauses};
4  reduce_nesting({'case', Head, [CaseClause|OuterCaseClauses]}) ->
5      {'case', Head, NewCaseClauses} =
6        reduce_nesting({'case', Head, OuterCaseClauses}),
7      {'case', Head, [CaseClause|NewCaseClauses]}.
8
9  if_in_case({OuterPattern, OuterGuard, [{'if', InnerClauses}]}) ->
10   [{OuterPattern, booth_true(OuterGuard, InnerGuard), InnerBody}
11     || {InnerGuard, InnerBody} <- InnerClauses].
```

This algorithm (Listings 4.2) finds the first lift-able `if` expression nested in a `case` expression and then it connects the `case` guard with all of the `if` guards using the `both_true` function (Described in Section 4.1).

Note that there are requirements of applicability for preserving the semantical meaning of the code, and the code quality.

The first such precondition is the `if`, as a single expression shall be the body of a `case` clause. The second requirement is that the `if` statement must always have at least one guard evaluating to `true`. The reasons are identical to the `if_in_if` (Section 4.1) transformation.

On Table 4.2 we demonstrate an example for lifting an `if` from a `case` -expression.

```
if_in_case_from(X, Y, Z) ->
    case X of
        pattern1 ->
            if
                Y + Z == 10 ->
                    expr1;
                true ->
                    expr2
            end;
        pattern2 ->
            expr3
    end.
```

```
if_in_case_to(X, Y, Z) ->
    case X of
        pattern1 when Y + Z == 10 ->
            expr1;
        pattern1 ->
            expr2;
        pattern2 ->
            expr3
    end.
```

Table 4.2: An example for lifting an if expression nested in a case expression

4.3 case in case

Probably, this is the most compound case of all. We have to consider not only patterns but also the guards, the head expressions of the `case` expressions and variables bound in the patterns.

Listing 4.3: An algorithm to lift a `case` expression from another

```
1  reduce_nesting({'case', OuterHead,
2    [CaseClause = {_, _, [{'case', _, _}]} | OuterCaseClauses]}) ->
3    {'case', NewHead, NewClauses} =
4      case_in_case(OuterHead, CaseClause),
5    {'case', NewHead, NewClauses ++
6      [{make_tuple(P, joker()), G, B}
7        || {P, G, B} <- OuterCaseClauses]};
8  reduce_nesting({'case', OuterHead,
9    [{Pattern, Guard, Body} | OuterCaseClauses]}) ->
10   {'case', NewHead, NewCaseClauses} =
11     reduce_nesting({'case', OuterHead, OuterCaseClauses}),
12   {'case', NewHead,
13     [{make_tuple(Pattern, joker()), Guard, Body} |NewCaseClauses]}.
14
15 case_in_case(OuterHead, {OuterPattern, OuterGuard,
16   [{'case', InnerHead, InnerClauses}]}) ->
```

```
17    NewHeads = make_tuple(OuterHead, InnerHead),
18    NewClauses =
19      [{make_tuple(OuterPattern, InnerPattern),
20        booth_true(OuterGuard, InnerGuard), InnerBody}
21        || {InnerPattern, InnerGuard, InnerBody} <- InnerClauses],
22    {'case', NewHeads, NewClauses}.
```

This algorithm (Listings 4.3) finds the first lift-able `case` expression nested in another and then it connects the outer guard and pattern with all of the inner guards and patterns using the `booth_true` (Described in Section 4.1) and the `make_tuple` functions. The `make_tuple` function creates the representation of a tuple from two element. We use the `joker()` function furthermore, that creates the representation of a wildcard pattern.

Note that there are conditions of applicability for preserving the semantical meaning of the code, and for preserving the code quality.

The first such precondition is the inner `case`, as a single expression shall be the body of an outer `case` clause.

The second requirement is that the `case` statement must always have at least one clause matching. It is needed because if non of the inner clauses match then Erlang throws an `exception error: no case clause matching (the head)`, and recreating it is hard, unpleasant and unreadable. This precondition is not checked in the algorithm, but could be in the `case_in_case` function.

Last, but not least, the inner `case`'s head should never cause any side effect (including errors). This is required, because if the pattern above the inner `case` did not match, the inner head would not cause side effect, however, after applying this refactoring, it would behave differently.

On Table 4.3 we demonstrate an example for lifting a `case` expression from an other.

```
case_in_case2_from(X, Y, Z) ->
   case X of
       pattern1 ->
           case Y of
               pattern2 when Z == 1 ->
                   expr1;
               pattern3 ->
                   expr2;
               _ ->
                   expr3
           end;
       _ ->
           expr4
   end.
```

```
case_in_case2_to(X, Y, Z) ->
   case {X, Y} of
       {pattern1, pattern2} when Z == 1 ->
           expr1;
       {pattern1, pattern3} ->
           expr2;
       {pattern1, _} ->
           expr3;
       _ ->
           expr4
   end.
```

Table 4.3: An example for lifting a case expression nested in an other

4.4 case in if

Listing 4.4: An algorithm to lift a `case` expression from an `if` expression

```
1  reduce_nesting({'if',
2    [CaseClause = {_, [{'case', _, _}]}|OuterIfClauses]}) ->
3      {Head, CaseClauses} = case_in_if(CaseClause),
4      TransformedIfClauses = [{joker(), IfGuard, IfBody}
5       || {IfGuard, IfBody} <- OuterIfClauses],
6      {'case', Head, CaseClauses ++ TransformedIfClauses};
7  reduce_nesting({'if', [{IfGuard, IfBody}|OuterIfClauses]}) ->
8      {'case', Head, NewCaseClauses} =
9        reduce_nesting({'if', OuterIfClauses}),
10     {'case', Head, [{joker(), IfGuard, IfBody}|NewCaseClauses]}.
11
12 case_in_if({OuterGuard, [{'case', Head, InnerClauses}]}) ->
13   {Head,
14     [{InnerPattern, booth_true(OuterGuard, InnerGuard), InnerBody}
15       || {InnerPattern, InnerGuard, InnerBody} <- InnerClauses]}.
```

This algorithm (Listings 4.4) finds the first lift-able `case` expression nested in an `if` expression. Because otherwise we can not do the refactoring, we transform the `if` expression into a `case` expression using the inner `case`'s head and joker or otherwise known as wildcard patterns. Finally, we lift the inner `case` expression by connecting the outer guard with all of the inner guards using the `booth_true` (Described in Section 4.1) functions.

Note that there are conditions of applicability for preserving the semantical meaning of the code, and for preserving the code quality.

The first such precondition is the inner `case`, as a single expression shall be the body of an `if` clause.

The second requirement is that the `case` statement must always have at least one clause matching.

Lastly, the inner `case`'s head should never case any side effect (including errors).

The reasons are identical to the `case_in_case` (Section 4.3) transformation.

On Table 4.4 we demonstrate an example for lifting a `case` from an `if` -expression.

```
case_in_if_form(X, Y, Z) ->
    if
        Y + Z == 10 ->
            case X of
                pattern1 ->
                    expr1;
                _ ->
                    expr2
            end;
        true ->
            expr3
    end.
```

```
case_in_if_to(X, Y, Z) ->
    case X of
        pattern1 when Y + Z == 10 ->
            expr1;
        _ when Y + Z == 10 ->
            expr2;
        _ ->
            expr3
    end.
```

Table 4.4: An example for lifting a case expression nested in an if expression

4.5 `case in function`

Listing 4.5: An algorithm to lift a `case` expression from function

```
1 reduce_nesting({Fun = {'fun', _},
2   [CaseClause = {_, [{'case', _, _}]} | OuterFunClauses]}) ->
3   {Fun, case_in_fun(CaseClause) ++ OuterFunClauses};
4 reduce_nesting({Fun = {'fun', _}, [FunClause |OuterFunClauses]}) ->
5   {Fun, NewFunClauses} = reduce_nesting({Fun, OuterFunClauses}),
6   {Fun, [FunClause | NewFunClauses]}.
7
8 case_in_fun({OuterPatterns, OuterGuard,
9   [{'case', Head, InnerClauses}]}) ->
```

```
10    [{boot_match(OuterPatterns, {Head, InnerPattern}),
11      booth_true(OuterGuard, InnerGuard), InnerBody}
12      || {InnerPattern, InnerGuard, InnerBody} <- InnerClauses].
```

This algorithm (Listings 4.5) finds the first lift-able `case` expression nested in a function and then it connects the function guard and patterns with all of the inner `case` guards and patterns using the `booth_true` (Described in Section 4.1) and the `both_match` functions. The `both_match` function finds a pattern in a list of patterns, and connects the found and the given pattern using an as-pattern.

Note that there are conditions of applicability for preserving the semantical meaning of the code, and for preserving the code quality.

The first such precondition is the inner `case`, as a single expression shall be the body of a function clause.

The second requirement is that the `case` statement must always have at least one clause matching.

The reasons are identical to the `case_in_case` (Section 4.3) transformation.

Lastly, the inner `case`'s head should be a variable variable bound in the function patterns. It needed to be able to replace the `case` patterns in the function patterns.

On Table 4.5 we demonstrate an example for lifting a `case` expression from a function.

```
case_in_func_from(pattern1, Y, Z) ->
   case Y of
       pattern2 when Z == 1 ->
           expr1;
       pattern3 ->
           expr2;
       _ ->
           expr3
   end;
case_in_func_from(pattern2, Y, Z) ->
   expr4.
```

```
case_in_func_to(pattern1, pattern2, Z) when Z == 1 ->
   expr1;
case_in_func_to(pattern1, pattern3, Z) ->
   expr1;
case_in_func_to(pattern1, _, Z) ->
   expr3;
case_in_func_to(pattern2, Y, Z) ->
   expr4.
```

Table 4.5: An example for lifting a case expression from a function

4.6 `if` in function

Listing 4.6: An algorithm to lift an `if` expression from function

```
reduce_nesting({Fun = {'fun', _},
  [IfClause = {_, _, [{'if', _}]}|OuterFunClauses]}) ->
    {Fun, if_in_fun(IfClause) ++ OuterFunClauses};
reduce_nesting({Fun = {'fun', _}, [FunClause|OuterFunClauses]}) ->
    {Fun, NewFunClauses} = reduce_nesting({Fun, OuterFunClauses}),
    {Fun, [FunClause|NewFunClauses]}.

if_in_fun({OuterPatterns, OuterGuard, [{'if', InnerClauses}]}) ->
  [{OuterPatterns, booth_true(OuterGuard, InnerGuard), InnerBody}
    || {InnerGuard, InnerBody} <- InnerClauses].
```

This algorithm (Listings 4.6) finds the first lift-able `if` expression nested in a function and then it connects the function guard with all of the `if` guards using the `booth_true` function (Described in Section 4.1).

Note that there are requirements of applicability for preserving the semantical meaning of the code, and the code quality.

The first such precondition is the `if`, as a single expression shall be the body of a function clause. The second requirement is that the `if` statement must always have at least one guard evaluating to `true`. The reasons are identical to the `if_in_if` (Section 4.1) transformation.

On Table 4.6 we demonstrate an example for lifting an `if` expression from a function.

```
if_in_func_from(pattern1, Y, Z) ->
   if
       Y + Z == 10 ->
           expr1;
       true ->
           expr2
   end;
if_in_func_from(pattern2, _, _) ->
   expr3.
```

```
if_in_func_to(pattern1, Y, Z) when Y + Z == 10 ->
   expr1;
if_in_func_to(pattern1, Y, Z) ->
   expr2;
if_in_func_to(pattern2, _, _) ->
   expr3.
```

Table 4.6: An example for lifting an if expression inside a function

4.7 The united algorithm

Listing 4.7: A combined algorithm to lift an `if` or a `case` expression

```
%% something in an if
reduce_nesting({'if',
  [IfClause = {_, [{'if', _}]} | OuterIfClauses]}) ->
  {'if', if_in_if(IfClause) ++ OuterIfClauses};
reduce_nesting({'if',
  [CaseClause = {_, [{'case', _, _}]} | OuterIfClauses]}) ->
  {Head, CaseClauses} = case_in_if(CaseClause),
  TransformedIfClauses = [{joker(), IfGuard, IfBody}
    || {IfGuard, IfBody} <- OuterIfClauses],
  {'case', Head, CaseClauses ++ TransformedIfClauses};
reduce_nesting({'if',
  [IfClause = {IfGuard, IfBody} | OuterIfClauses]}) ->
  case reduce_nesting({'if', OuterIfClauses}) of
    {'case', Head, NewCaseClauses} -> % case_in_if
      {'case', Head, [{joker(), IfGuard, IfBody} |NewCaseClauses]};
    {'if', NewIfClauses} -> %if_in_if
      {'if', [IfClause | NewIfClauses]}
  end;

%% something in a case
reduce_nesting({'case', OuterHead,
  [CaseClause = {_, _, [{'case', _, _}]} | OuterCaseClauses]}) ->
  {'case', NewHead, NewClauses} =
    case_in_case(OuterHead, CaseClause),
  {'case', NewHead, NewClauses ++
    [{make_tuple(P, joker()), G, B}
      || {P, G, B} <- OuterCaseClauses]};
reduce_nesting({'case', Head,
  [CaseClause = {_, _, [{'if', _}]} | OuterCaseClauses]}) ->
  {'case', Head, if_in_case(CaseClause) ++ OuterCaseClauses};
reduce_nesting({'case', OuterHead,
  [CaseClause = {Pattern, Guard, Body} | OuterCaseClauses]}) ->
  case reduce_nesting({'case', OuterHead, OuterCaseClauses}) of
    {'case', OuterHead, NewCaseClauses} -> %if_in_case
      {'case', OuterHead, [CaseClause | NewCaseClauses]};
    {'case', NewHead, NewCaseClauses} -> %case_in_case
      {'case', NewHead,
        [{make_tuple(Pattern, joker()), Guard, Body}
          | NewCaseClauses]}
  end;

%% something in a function
reduce_nesting({Fun = {'fun', _},
  [IfClause = {_, _, [{'if', _}]} | OuterFunClauses]}) ->
  {Fun, if_in_fun(IfClause) ++ OuterFunClauses};
```

```
46 reduce_nesting({Fun = {'fun', _},
47   [CaseClause = {_, [{'case', _, _}]} | OuterFunClauses]}) ->
48   {Fun, case_in_fun(CaseClause) ++ OuterFunClauses};
49 reduce_nesting({Fun = {'fun', _}, [FunClause |OuterFunClauses]}) ->
50   {Fun, NewFunClauses} = reduce_nesting({Fun, OuterFunClauses}),
51   {Fun, [FunClause | NewFunClauses]}.
```

Finally, we give an algorithm for any case of lifting that can be seen on Listings 4.7.

The algorithms can be joint quite easily. First of all, anytime, we find a special case, we call the corresponding transformer function. (Note, that the helper or transformer functions are defined in the corresponding sections of this chapter) If we did not find a special case, dependently on what the recursive call might return, we use a `case` expression to join the result of the recursive call, and the current clause. When nessecary, we transform the current clause into a `case` or a function caluse.

Chapter 5

Eliminating `hd` and `tl` calls

We have discussed how one can eliminate the usage of guards and lift nested expressions. Now, we introduce our last refactoring, that can be used to eliminate calls to the `hd` or `tl` functions.

We find this refactoring justifiable since the usage of these functions are generally not suggested when calling it on a variable, and one can use as-pattern alternatively. The usage of as-pattern results in a more elegant, and concise code, furthermore, we can eliminate code-duplication if the call refers to the same variable multiple times.

5.1 The Transformation

This transformation is the least sophisticated of all; still, we need to consider many different cases. The idea is that instead of using `hd` or `tl` calls, one can bind a variable to the head or tail of a list using pattern matching. Moreover, we can do this at the line of the assignment using as-pattern. After binding a variable to the head or tail of the list, we can exchange the `hd` or `tl` calls everywhere. We may choose any available names for the new variables, but for simplicity, our implementation uses the original name extended with a `_Hd` or `_Tl` post-fix. Later on, the user can choose a more descriptive name and rename it.

5.1.1 Conditions of applicability

We intuitively call this a "transformation", and not a "refactoring", because we can not ensure semantic equivalence. For the simplest example, if the `hd` function is called on an empty list, different errors will be thrown at different places. The same stands for any bad argument in the `hd` or `tl` calls. Ergo, if the original code was written so that the `hd` or `tl` call may throw an error, the refactored code will behave differently.

Nevertheless, we would like to preserve the meaning of the code as much as possible; therefore,

the transformation shall only be applied when the code meets some conditions.

Firstly, the `hd` or `tl` function should be used on a variable. It is needed because we want to use as-pattern at an already existing assignment. If the `hd` or `tl` call on the return value of an expression, we would not want to change anything.

An example for such case is shown in Figure 5.1

```
    end,
hd(cerl_trees:fold(Fun, [], Tree)).
```

Figure 5.1: An example for a non-transformable `hd` call

Secondly, if there are guards between the assignment and the use of the `hd` or `tl` function, we reject the transformation. This stands for the case as well when the `hd` or `tl` function is placed in a guard. We require this since the guards can separate different cases, and these calls can be used after some specific checks. If that were the case, refactoring would cause the change of the code's meaning. Moreover, the `hd` or `tl` call can be located in a guard. In that case, calling them on a bad argument would not throw an error because of how erlang manages errors in guards.

An example for such case is shown in Figure 5.2

```
output(Level, StrOrFun, Sender, From, St) when is_integer(Level) ->
    case selected_by_level(Level, stdout, St) of
    true when hd(StrOrFun) == "$tc_html" ->
        output(stdout, tl(StrOrFun), Sender, From, St);
    true when is_function(StrOrFun) ->
        output(stdout, StrOrFun(stdout), Sender, From, St);
    true ->
        output(stdout, StrOrFun, Sender, From, St);
    false ->
        ok
    end,
```

Figure 5.2: An other example for a non-transformable `hd` call

Chapter 6

Validation

Validation is the process of making something officially acceptable or approved. We find it essential to validate our refactorings using data we extract from real-world code-bases. In the following chapter, we will demonstrate some measures that can verify the the introduced refactorings.

6.1 Real life applicability

To demonstrate where the defined refactorings could be applied and how some it can change the code, I will present a couple of examples.

6.1.1 Transforming guard to pattern

After I wrote the refactoring, that can eliminate the calls to the function `length` I was concerned, no one would make such mistakes. I thought so because I have been thinking about this problem from quite a while and I could not even imagine little more experienced Erlang developers could ever even use `if` expressions.

Driven from this disappointment, I was exasperatedly looking for examples where my freshly implemented refactoring could be useful. My first try was on GitHub, which unfortunately did not offer to search for regular expressions that made me look up numerous pages of projects with the given specifications.

After realizing this will not work, I had the idea of using source-code on my computer. Basically, two sources could come into question that was large enough to have a chance of finding the bothersome pattern: The RefactorErl Project and the source of Erlang/OTP.

I used the following command to recursively searches for the specified pattern from a directory:

```
grep -r " length(.*) [>=]*[:/]*[=<] [0-5] "
```

It will find all calls to function `length` when the result of that call is compared with a Number between zero and five in the source code.

For greatest surprise in both The RefactorErl Project and the source of Erlang/OTP, I found many occurrences of the given pattern, 24 in the RefactorErl and 48 in the Erlang/OTP. In both cases, numerous matches could be eliminated by the newly defined refactoring, improving the quality of the source code.

The results can be seen on Figure 6.2 and on Figure 6.1.

```
boldi@boldi-aspire-e5-573:/usr/lib/erlang/lib$ grep -r " length(.*) [>=]*[:/]*[=<] [0-5] "
kernel-6.5/src/inet_res.erl:                  length(Msg#dns_rec.qdlist) =/= 1 ->
kernel-6.5/src/erl_epmd.erl:            Data when length(Data) < 4 ->
kernel-6.5/src/erl_epmd.erl:            Data when length(Data) < 2 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_prev3b.erl:_ = [if length(Comp) =:= 1 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_prev3c.erl:_ = [if length(Comp) =:= 1 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_prev3a.erl:_ = [if length(Comp) =:= 1 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_v3.erl:_ = [if length(Comp) =:= 1 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_v2.erl:_ = [if length(Comp) =:= 1 ->
megaco-3.18.6/src/binary/megaco_per_media_gateway_control_v1.erl:_ = [if length(Comp) =:= 1 ->
asn1-5.0.9/src/asn1ct_imm.erl:    %%    _ = [if length(Comp) =:= 4; byte_size(Comp) =:= 4 -> [] end ||
hipe-3.19.1/cerl/erl_bif_types.erl:                List when length(List) >= 2 ->
hipe-3.19.1/cerl/erl_bif_types.erl:              false when length(Arity) =:= 1 ->
hipe-3.19.1/cerl/erl_bif_types.erl:              true when length(Arity) =:= 1 ->
hipe-3.19.1/cerl/erl_bif_types.erl:                   Stypes when length(Stypes) >= 1 ->
hipe-3.19.1/cerl/erl_bif_types.erl:                 List when length(List) >= 1 ->
hipe-3.19.1/cerl/cerl_lib.erl:                    var when length(Vs) =:= 1 ->
hipe-3.19.1/icode/hipe_icode_ssa_struct_reuse.erl:    %#call{type = primop, dstlist = List} when length(List) >= 1 -
> struct;
stdlib-3.10/src/erl_eval.erl:     Bs0, Lf, Ef, RBs) when length(As0) =< 1 ->
stdlib-3.10/src/erl_eval.erl:     Bs, Lf, Ef, RBs) when length(As0) =< 1 ->
stdlib-3.10/src/uri_string.erl:get_separator(L) when length(L) =:= 0 ->
stdlib-3.10/src/qlc_pt.erl:                    length(VarValues) =< 1 andalso
stdlib-3.10/src/qlc_pt.erl:    {join_gens2(lists:filter(fun(C) -> length(C) =:= 2 end, Cs), FD, Skip),
stdlib-3.10/src/qlc_pt.erl:                case length(GIds) =< 1 of
stdlib-3.10/src/qlc_pt.erl:qlcmf(?QLC_Q(L1, L2, L3, L4, LC0, Os0), F, Imp, A0, No0) when length(Os0) < 2 ->
stdlib-3.10/src/qlc_pt.erl:qlcmf(?IMP_Q(L1, L2, LC0, Os0), F, Imp=true, A0, No0) when length(Os0) < 2 ->
stdlib-3.10/src/qlc.erl:        length(L) < 3 andalso length(Stacktrace) =:= D ->
stdlib-3.10/src/erl_lint.erl:        {tuple, _, Exprs} when length(Exprs) =/= 2 -> false;
snmp-5.4.3/src/misc/snmp_misc.erl:%%      length(TMask) == 0 | length(TAddr1)
snmp-5.4.3/examples/ex1/ex1.erl:                                  length(Cols) == 3 ->
common_test-1.18/src/ct_groups.erl:    Where = if length(Known) == 0 ->
common_test-1.18/src/ct_cover.erl:                      Res when Res == [] ; length(Res) == 1 -> % 1 app = ok
xmerl-1.3.22/src/xmerl_scan.erl:                   Delim,_Acc,PEName,_NS,PENesting) when length(PENesting) /= 0 ->
xmerl-1.3.22/src/xmerl_xsd_type.erl:    L when length(L) == 1; length(L) == 2 ->
xmerl-1.3.22/src/xmerl_xsd.erl:              L when length(L) =< 1 -> % Part1:3.11.4.3
compiler-7.4.8/src/beam_ssa_codegen.erl:            '-' when length(Args) =:= 2 -> fsub;
compiler-7.4.8/src/sys_core_fold.erl:eval_append(Call, #c_literal{val=Cs}, List) when length(Cs) =< 4 ->
compiler-7.4.8/src/sys_core_fold.erl:  when length(MoreArgs0) >= 0 ->
compiler-7.4.8/src/core_pp.erl:    length(Es) < 4 andalso lists:all(fun is_simple_term/1, Es);
dialyzer-4.1/src/dialyzer_options.erl:assert_filenames_form(Term, [FileName|Left]) when length(FileName) >= 0 ->
dialyzer-4.1/src/dialyzer_options.erl:assert_filename(FileName) when length(FileName) >= 0 ->
tools-3.2.1/src/xref_utils.erl:            when length(AllowedValues) =:= 1 ->
syntax_tools-2.2.1/src/igor.erl:    Attrs = if length(Env#merge.sources) =:= 1 ->
syntax_tools-2.2.1/src/igor.erl:            if length(Env#merge.sources) =:= 1 ->
public_key-1.7/src/public_key.erl:verify_hostname_match_default0({ip,R}, {iPAddress,P}) when length(P) == 4 ->
ssl-9.4/src/ssl_certificate.erl:        {self, _} when length(Path) == 1 ->
ssl-9.4/src/ssl_logger.erl:        H when length(H) < 2 ->
edoc-0.11/src/edoc_parser.yrl:  if length(element(1, '$1')) == 1 ->
edoc-0.11/src/edoc_parser.yrl:     length(element(1, '$1')) == 0 ->
```

Figure 6.1: Found possibly transformable expressions in the Erlang/OTP

Figure 6.2: Found possibly transformable expressions in The RefactorErl Project

6.1.2 `hd` or `tl` elimination

Similarly, as in the previous section, the usage of `hd` and `tl` functions can be grepped easily.

```
grep -r " hd([a-Z]*)"
grep -r " tl([a-Z]*)"
```

The first grep resulted in 120 mathces, and the second in 40 matches in the Erlang source code.

6.1.3 Branch lift

Writing a grep for this refactoring is impossible. Instead, I wrote a query in the RefactorErl tool, that finds all the applicable instances. Furthermore, all of the pre-conditions could be implemented in this query, thus only the applicable results were selected.

There are some drawback of this method compared to the grep. Firstly, it takes much more time. Secondly, to run the query, we need to add the source code first. Since the RefactorErl needs to process all the data it is added, it was very slow, and only a few modules were added.

Still, in these few modules more than a hundred instances of applicability were found.

6.2 Metrics

Software metrics are a standard of measure of a degree to which a software system possesses some property. They are used widely and have become an integral part of the software development process. In this section, we present some representatives of different metric families.

6.2.1 Cyclomatic Complexity

One of the most well-known metrics is Thomas J. McCabe's Cyclomatic Complexity (CC). It is based on the control flow graph (CFG) of the program, and it gives the number of linearly independent paths through a structured program. [19]

The Cyclomatic Complexity is defined as

$$CC = E - N + 2P$$

where N is the number of vertices, E is the number of edges, and P is the number of connected components of the CFG.

Since our refactorings will not change the linearly independent paths through the program, the CC should not change its value. Therefore, we will not use this metric in our measurements.

6.2.2 Halstead Complexity Measures

Other very well known metrics are Maurice Howard Halstead's complexity measures. They are based on the number of operators and operands in the program. [20]

To calculate them, let η_1 equal the number of distinct operators and η_2 equal the number of distinct operands. Furthermore, let N_1 equal the total number of operators and N_2 equal the total number operands.

After introducing these numbers, there are three measures we would like to discuss. The first is the Halstead volume (V) which expresses the amount of information to comprehend in order to understand the program. It can be calculated using the following formula:

$$V = (\eta_1 + \eta_2) \cdot \log_2(N_1 + N_2)$$

The second measure is the difficulty (D) of a program. It aims to express how difficult it is to understand the program and can be calculated using the following formula:

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

The last one is the Halstead effort (E) that expresses how much effort would it take to rewrite the code. It can be calculated by the product of the previous measures:

$$E = D \cdot V$$

6.2.3 Cognitive Functional Size

A newer metric is the Cognitive Functional Size (CFS) introduced by Yingxu Wang. It bases on the cognitive weight of different basic control structures which can be seen on Table 6.1.[21]

Category	Basic Control Structure	Cognitive Weight
Sequence	Sequence	1
Branch	If	2
	Case	3
Embedded component	Function call	2
	Recursion	3
Concurrency	Parallel	4
	Interrupt	4

Table 6.1: The cognitive weight of different basic control structures

We defined the cognitive weight of a software component (W_c) as the sum of the cognitive weights of its q linear blocks. Since each block is built of m a nested layers, and these layers can contain n independent linear block, the total cognitive weight of W_c can be calculated using the following formula:

$$W_c = \sum_{j=1}^{q} \prod_{k=1}^{m} \sum_{i=1}^{n} (\text{The cognitive weight of i-th component at the k-th nesting layer in the q-th block})$$

6.2.4 Unique Complexity Metric

Lastly, we present the unique complexity metric introduced by Sanjay Misra. It is a hybrid metrics that combines the Halstead and the cognitive metrics. [22]

To calculate it, let $N^{(k)} = N_1^{(k)} + N_2^{(k)}$ where $N_1^{(k)}$ and $N_2^{(k)}$ are the total number of operators and operands in the k-th line. Moreover, let $CW^{(k)}$ equal the cognitive weight of the k-th line. The unique complexity metric can be calculated using the following formula.

$$UCM = \sum_{i=1}^{n} 1 + N^{(i)} \cdot CW^{(i)}$$

6.3 Measurements

Of course, none of these measures can represent the quality of a program alone. There other hybrid measures, such as the Maintainability Index (MI)[23], but we only use the above-described

ones.

In this section, we present measurement data for each defined refactoring. We randomly chose real-life instances of applicabilities from the Erlang source code, mostly from the mnesia, edoc and the parser modules, but other modules were used as well. Then we measured the value of the described metrics for booth the original, and the transformed code.

We wanted to present the growth or decrease in the measure intuitively, so first off, we started with the difference of the original, and the refactored code's metrics. The problem with this is that when we use a more significant component, the measures are big, and the differences are also much higher. It does not represent well, how much better or worse the code might have gotten.

After this, we considered the ratio of the original and the transformed code. It represents the ratios well, but its representation is advantageous in a coordinate system, and we would need one for each metric that would need too much space.

Lastly, we chose the logarithm of the ratio of the original and the transformed code. It represents the ratios well, and one can see the increases and decreases of the metrics intuitively.

How the code was found?

All used code and their transformed form can be accessed on the following GitHub repository:
`https://github.com/boldar99/RefErl_Measures`
Furthermore, the visualisations and the measurement data can be accessed through the following link:
`https://public.tableau.com/profile/po.r.boldizs.r#!/vizhome/MetricVisualisation/Measures`.

These links will be valid at least until 2021, 1st of May.

6.3.1 Branch lifts

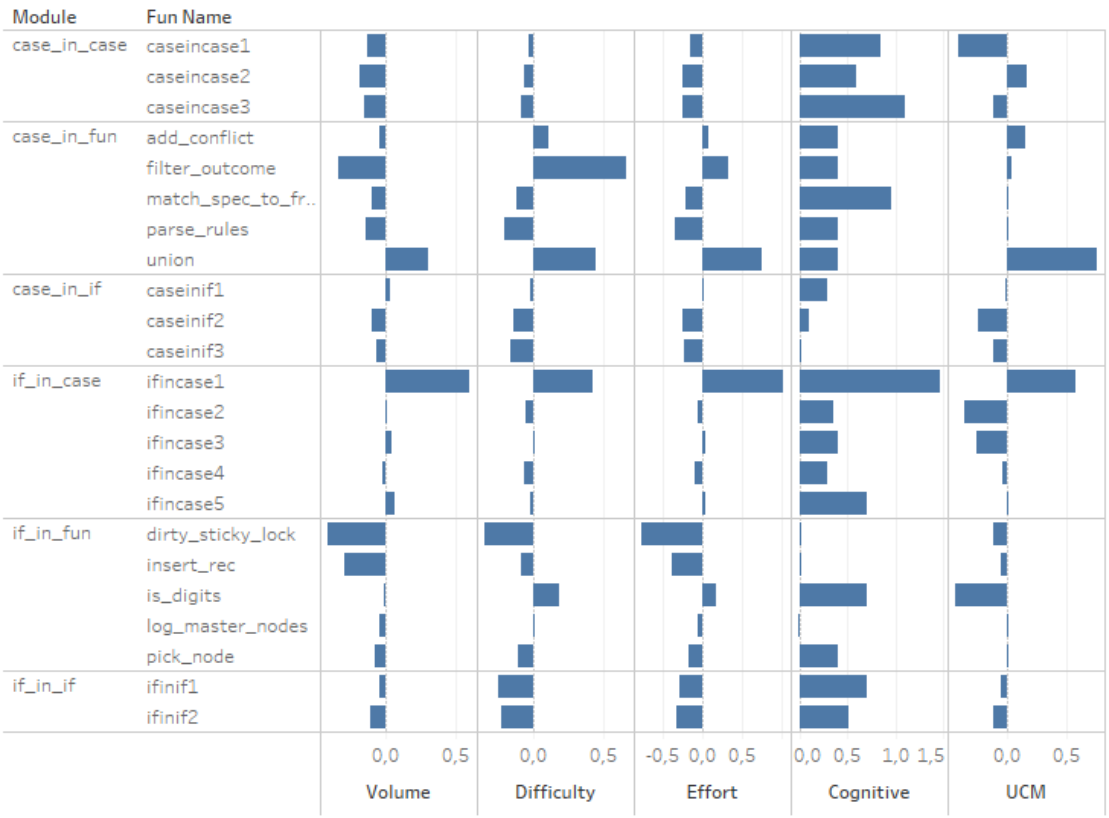

Figure 6.3: Logarithm of the ratio in Branch Lift

The visualisation of the measures can be seen on Figure 6.3.

It is very promising that the cognitive functional size almost always increases significantly. This measure highly depends on the structure and the nesting of a function, so these increases were expected.

The Halstead measures decrease a little bit most of the time, but sometimes increases can be observed as well. The Halstead measures bases on the number of operators and operands without the consideration of the inner structure of the code. Taken this into consideration, we expected these measures, since the branch lifting refactoring eliminates nesting, thus restructures code at the price of creating little code duplication.

Lastly, the UCM did not performed as firstly expected. This was due to the miss-understanding of how it works. Apparently, the UCM treats the source-code as set of lines, without the consideration of nesting, or inner structure. After this realisation, it was clear, why the UCM produced such data.

6.3.2 `length` elimination

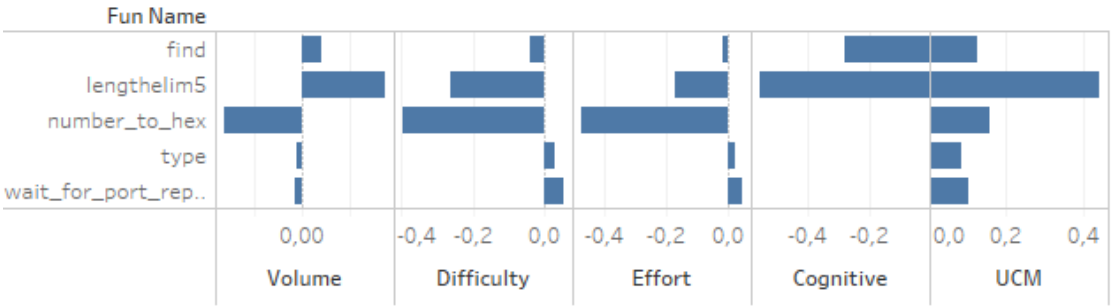

Figure 6.4: Logarithm of the ratio in `length` elimination

The visualisation of the measures can be seen on Figure 6.4. Unfortunately, the Cognitive Functional Size of the `wait_for_port_reply` and the `type` functions are partially defined only, as a result their logarithm of ratio is 0.

When eliminating a `length` call, we express the same information using a pattern matching. This way, we may use more operators than originally, but we may not. As a result, the Halstead measures vary strongly.

The cognitive functional sizes decrease every time, since the cognitive weight of a `case` expression is higher then an `if` expression.

Lastly, the refactoring seem to affect the UCM very positively. Only increases can be seen throughout the measures.

6.3.3 `hd` or `tl` elimination

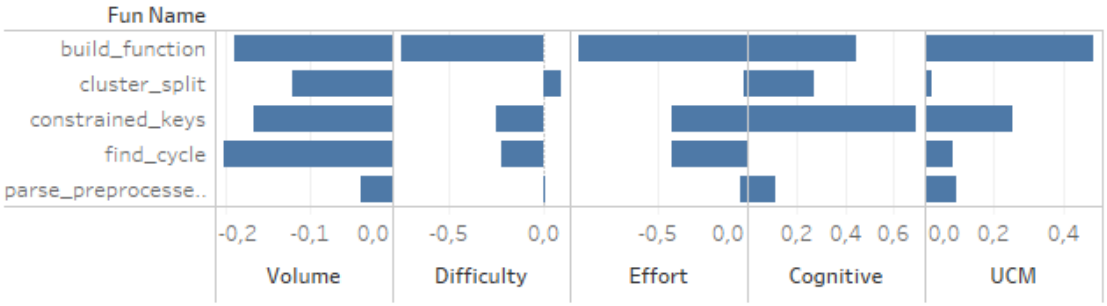

Figure 6.5: Logarithm of the ratio in `hd` or `tl` elimination

The visualisation of the measures can be seen on Figure 6.5.

Since the `hd` and `tl` refactoring increases the number of operators in the code, the Halstead measures perform weakly.

On the other hand, we decrease the complexity of other expressions. Typically these calls were placed in an other function call, thus, booth the cognitive functional size and the UCM decreased.

Chapter 7

Conclusions

In my research, I was working on defining refactorings that can eliminate imperative-like patterns from Erlang source code. It is necessary because the influence of the imperative mentality can frequently be seen in Erlang programs.

7.1 Summary

In this paper I presented three refactoring methods.

The first one was transforming guard to pattern which eliminates redundant calls to the `length` function and furthermore transforms an `if` expression into a `case` expression. When this refactoring is applied, the execution time of the program will be improved without any semantic change in it.

The second was the branch lifting, for which I defined the steps and conditions of application for all possible cases of containment. The application of this refactoring can improve readability.

Thirdly, I presented the transformation, that can eliminate the calls to `hd` and `tl`.

I presented where the refactorings could be applied and gave a method for looking for instances for two of them, and implemented a query, that can find the last one. Furthermore, I measured how each transformation affects different software complexity metrics, and created visualisations from the results.

Moreover, I implemented these refactorings in the RefactorErl framework.

Lastly, I presented these ideas in the 3in Carpathian Basin Conference on Software and Security, 2019.

7.2 Future work

The main goal of this project was to define a clean-up methodology that transforms imperatively styled Erlang source-code into one that is more convenient to be read by functional programmers.

In defining this methodology, the verification of each implemented refactoring is important moreover we need to discover more imperative patterns that can be transformed.

7.2.1 Further verification

There are promising researches in the field of trustworthy refactorings [24]. In the future, I would like to investigate the proposed refactoring language and try to formalise my refactorings using this approach.

Hopefully, these efforts will make the future generation find the functional wold more attractive. Thus the friendly community of the functional languages could get bigger and more people would discover the beauty of composing a functional program.

Bibliography

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[2] Erlang introduction. `http://erlang.org/doc/system_architecture_intro/sys_arch_intro.html#id58791`. [Accessed: December 2019].

[3] Wikipedia: Erlang. `https://en.wikipedia.org/wiki/Erlang_(programming_language)`. [Accessed: December 2019].

[4] The erlang virtual machine: Beam. `https://blog.stenmans.org/theBeamBook/#CH-BEAM`. [Accessed: December 2019].

[5] Joe Armstrong. The development of erlang. *SIGPLAN Not.*, 32(8):196–203, August 1997.

[6] Refactorerl's website. `https://plc.inf.elte.hu/erlang/`. [Accessed: September 2019].

[7] Simon Thompson. Refactoring functional programs. In -, pages 331–357, 08 2004.

[8] Huiqing Li, Simon Thompson, George Orösz, and Melinda Tóth. Refactoring with wrangler, updated data and process refactorings, and integration with eclipse. In -, pages 61–72, 01 2008.

[9] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of erlang programs. In -, pages 13–24, 01 2009.

[10] Ackermann function. `https://en.wikipedia.org/wiki/Ackermann_function`. [Accessed: October 2019].

[11] Why are cons lists associated with functional programming? `https://softwareengineering.stackexchange.com/questions/132309/why-are-cons-lists-associated-with-functional-programming`. [Accessed: October 2019].

[12] Concise erlang. `https://www.cis.upenn.edu/~matuszek/ConciseGuides/ConciseErlang.html`. [Accessed: December 2019].

[13] Erlang coding standards & guidelines. `https://github.com/inaka/erlang_guidelines`. [Accessed: September 2019].

[14] What is the difference between == and =:= in erlang? https://stackoverflow.com/questions/9790815/what-is-the-difference-between-and-in-erlang-when-used-with-terms-in-gene. [Accessed: December 2019].

[15] Erlang: Boolean expressions. http://erlang.org/doc/reference_manual/expressions.html#boolean-expressions. [Accessed: September 2019].

[16] Side-effects in "if" guards. http://erlang.org/pipermail/erlang-questions/2008-May/035119.html. [Accessed: September 2019].

[17] Allowed functions in guards. https://en.wikibooks.org/wiki/Erlang_Programming/guards. [Accessed: September 2019].

[18] Erlang programming rules and conventions. http://www.erlang.se/doc/programming_rules.shtml. [Accessed: April 2020].

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[20] Maurice H. Halstead. Elements of software science. 1977.

[21] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74, 2003.

[22] Sanjay Misra and Ibrahim Akman. A unique complexity metric. In *International Conference on Computational Science and Its Applications*, pages 641–651. Springer, 2008.

[23] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, 1992.

[24] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. Towards trustworthy refactoring in erlang. *arXiv preprint arXiv:1607.02228*, 2016.

[25] Fred Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 2013.

[26] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[27] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[28] Ericsson AB. Erlang Programming Language. http://www.erlang.org. [Accessed: 2017.02.07].