The effect of EHOP on the writing of Program Analyzers

Brendan Mesters
Supervisor(s): Casper Bach Poulsen, Cas van der Rest
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Effect Handler Oriented Programming is a promising new programming paradigm, delivering separation of of concerns with regards to side effects in an otherwise functional environment. This paper discusses the applicability of this new paradigm to static code analysis programs. Different code analyzers often have many similar, if not identical pieces of code which could be abstracted away. This abstraction does not come natural to the programming paradigm of Functional Programming but are quite natural within EHOP. The current programming languages do not yet seem up to the task of rapid generalization of code and elimination of duplicate pieces of code. However, the concepts present in EHOP will almost certainly be able to eliminate much of this code reduction once the languages have matured further. The implicit passing of functionality will also allow for clearer code with less unnecessary visual clutter.

1 Introduction

The newly proposed programming paradigm of Effect Handler Oriented Programming (EHOP) is starting to see more practical research into its potential uses. EHOP languages look to bring a separation of concerns to the functional programming paradigm by introducing effects. Effects can do various things, they are, for example, used to handle the non-functional aspects of code, such as IO operations, state dependant operations and exceptions. This extra level of abstraction could be very useful for writing general code, as well as doing sand-boxing.

In an EHOP language you can define certain effects, just like you can define any other data type. An effect is very similar to an interface, in that it only gives you the type definitions of its functionality. Functions can require certain effects to be provided for them to run, these effects will then automatically be passed to any functions that this code calls. To provide an effect to a function you need to give implementations to all the functionality that an effect promises. In this way you are able to set up very generalized and adaptable code, all within an environment similar to Functional Programming.

A lot of research has already been done into the mathematical basis for EHOP in papers such as *Handlers of Algebraic Effects* by Gordon Plotkin and Matija Pretnar [1], which lays down the original mathematical concepts of Effect Handlers in computer science. *Type Directed Compilation of Row-Typed Algebraic Effects* by Daan Leijen [2] presents a research language, Koka [3], which uses row typed algebraic effects. Koka is also the language in which this research projects code has been written. And the article *Do Be Do Be Do* by Conor McBride et al. [4] explores a bidirectionally typed effect system which allows more effect variables to be omitted. There has however not yet been a lot of research into the difficulty of learning an EHOP language. Similarly there hasn't been much research into the application of EHOP on different types of programs.

This article focuses on the question of whether the concepts present in EHOP are of use when creating program analyzers. Furthermore the article will also function as a case study on how easy or difficult it is for someone with prior functional programming experience to learn an EHOP language.

For this, our main research question is: **How does the EHOP programming paradigm effect the writing of Program Analysis Tools**. This main research question is further subdivided into sub questions.

- **RQ1** Do the concepts present in EHOP translate well to program analysis tools?
- **RQ2** Does EHOP allow for more code reuse then otherwise possible?
- **RQ3** How does EHOP programming effect the readability of the code.
- **RQ4** How quickly can an EHOP language be picked up by someone with prior functional programming experience?

The contributions to the field that this paper aims to make are as follows.

- The paper has created a type-checker and an interpreter for Mini-ML [5] in an EHOP language, for which the implementation details can be found on the Github [1] and an explanation of the process can be found in Section 4.2
- It examines this code as to gain insights upon the different advantages and disadvantages that EHOP has. (Section 6)
- It provides a look into the learning process of EHOP programming, specifically the difficulties in using polymorphic effects. (Subsection 4.3 and Subsection RQ4in Section 6)
- This research shows pros and cons of the EHOP programming paradigm in the given domain of program analysis tools, as well as give us data on the ease of adoption of EHOP languages. (Section 6)

First Section 2 will briefly explain why EHOP might be in interesting programming paradigm for code analysis tools. Following this, the methods used in this paper will be discussed from a theoretical point of view in Section 3. Afterwards an in depth look at the process of this research will be given in Section 4, followed by Section 5 which will discuss the analysis methods. The results will be discussed in Section 6. Sections 7 and 8 will put a critical eye to the work presented in this paper. And lastly there will be Section 9 in which we will briefly show the final conclusions as well as suggest some possible future research in Section 10.

2 The advantages of EHOP for Code Analysis Tools

Static code analysis tools, as well as interpreters, often have very large parts of similar, and sometimes even identical code. In many of these cases this code is the structure of your program, the part that scans over the varying tokens from the

[1] https://github.com/BrendanMesters/Koka-Linter

```
Variable(name) ->
    if !pscope_contains(name)
    then
        add_error("Variable '" ++ name ++ "' was not in scope")
        ENull
    else
        match pget(name)
            Direct_value(v) -> v
            Actual_closure(e, clos) ->
                val old_scope = pget_state()
                pset_state(clos)
                val retval = e.type_checker()
                pset_state(old_scope)
                retval
```

```
Variable    (name: string) ->
    if !pscope_contains(name)
    then
        add_error("Variable '" ++ name ++ "' was not in scope")
        Null
    else
        match pget(name)
            Direct_value(v) -> v
            Actual_closure(e, clos) ->
                val old_scope = pget_state()
                pset_state(clos)
                val retval = e.test_substitute([name])
                pset_state(old_scope)
                retval
```

Figure 1: "Similarities in variable handling code in the Type Checker and Interpreter respectively"

abstract language tree and does differing things depending on the encountered token, as well as your application. A type checker may simply check if the arguments of any function (including simple mathematical functions such as Plus, Minus, etc) are of the expected types (using recursive calls to itself) while an interpreter would have to evaluate the actual values of said function applications.

For these reasons it has been decided that a type checker and an interpreter will be used to analyze the effects of EHOP and to try and answer the research questions. These two code analysis tools have been chosen as they have rather similar functionality, thus they have some parts of duplicate code. Clearly visible in the usage of variables, shown in Figure 1. Further examples and in depth discussion will be provided in Section 4.

3 Methodology

The article tries to gain insights into the practical applicability of EHOP, both in general and in the context of static code analyzers. Therefore this research will have a 3rd years bachelor of computer science program *program analysis tools* for the functional toy language of *Mini-ML* [5], which is in essence a simply typed Lambda Calculus. This should help us in answering all four of the research sub questions. The programming will be done in the span of 10 weeks, in which this research paper will also have to be written, and the programming will be done in the EHOP language Koka [3].

Focusing on the full breadth of the research question would be infeasible, due to the fact that there are many different program analysis tools. Thus its been decided that this project will only focus on the programming of a subset of program analysis tools. The two programs that this paper uses for its research are a *type checker* and an *interpreter*, as they are relatively similar while still being complex. These two programs should give good insights into the possible abstractions that can be implemented within code analysis tools as a whole. The code also automatically fulfils certain functionalities such as the detection of variable usage before assignment.

After the creation of the type checker and interpreter the code will be assessed. This assessment has been done in part by analyzing the code myself and giving substantiated answers to the research questions. This will give conclusions on RQ 1 to 3, each substantiated with arguments and code references where applicable. RQ 4, on the other hand, will be answered more as a subjective case study of my own experiences.

It would be preferable if we could use less subjective measurements to evaluate the code quality, but sadly this is not possible. Almost all code quality analysis tools are ran by a program as opposed to calculated by hand, since this would be a lot of work for relatively little results. The problem here is that all the existing tools are either created for a specific language or require you to be able to translate your code into some internal structure that their tool uses, which would, due to time, be unrealistic for this project. On top of that there is even still the problem that these tools are not made for an Effect Handler Oriented language, which could throw off certain measurements.

One method which is sometimes employed in the field of Programming Languages research is peer evaluation. This project tried to use peer evaluation too, but the amount of responses was to small to include in a scientific paper. However, there seemed to be interesting information in the few responses we did get, thus we have added this as a possible future research but left it out of this paper.

4 The Implementation of the EHOP Code Analysis Tools

During the implementation of the interpreter and type checker many very similar patterns have been found. The code tries to abstract these away as much as possible, the example of the variable code (Figure 1) was already given in Section 2. For more detail on the reasons why EHOP should be interesting for static code analysis tools see Section 2. A few complex Mini-ML programs have also been created inside the `main.kk` file for the sake of testing the correctness of the written code.

The original aim of this paper was to create a type-checker and an interpreter that where abstracted enough that they did not contain any duplicate code anymore. During the process of creating this code several severe problems where discovered. The following section will discuss the full original concept in Sub-Section 4.1, the actual implementation code, highlighting interesting parts in Sub-Section 4.2, and lastly it will discuss the issues that where encountered during the programming in Sub-Section 4.3.

4.1 Original Plan of the Project

The original plan was to create highly generalized code which could be transformed into a myriad of different code analysis

```
effect handle_expression<a>
    fun handle_Num(n: int): a
    fun handle_EFalse(): a
    fun handle_ETrue(): a
    fun handle_Variable(name: string): a
    fun handle_Lambda(arg: patern, body: expression): a
    fun handle_If(cond: expression, thus: expression, otherwise: expression): a
    fun handle_MLpair(e1: a, e2: a): a
    // Handle apply may have a name argument, this is only provided
    // if the function was aquired with a variable and will be the
    // variable name.
    fun handle_Lambda_Apply(arg: patern, arg_vals: expression, body: expression): a
    fun handle_Variable_Apply(var_name: string, arg_vals: expression): a
    fun handle_Let(arg: patern, arg_val: expression, body: expression): a
    fun handle_Letrec(arg: patern, arg_val: expression, body: expression): a
    fun handle_Equals(e1: expression, e2: expression): a
    fun handle_EMult(e1: expression, e2: expression): a
    fun handle_EAdd(e1: expression, e2: expression): a
    fun handle_ESub(e1: expression, e2: expression): a
    fun get_Null_val(): a
```

```
fun analyzer (inp: expression) :
<div, exn, errors-and-warnings, polymorphic_scope<expression>, handle_expression<a>, console|e> a
    match inp
        Num(n) -> handle_Num(n)
        EFalse -> handle_EFalse()
        ETrue -> handle_ETrue()
        Variable(name) -> handle_Variable(name)
        Lambda(arg, body) -> handle_Lambda(arg, body)
        If(c, t, e) -> handle_If(c, t, e)
        MLpair(e1, e2) -> handle_MLpair(e1.analyzer, e2.analyzer)
        Apply(func, arg_vals) ->
            match func
                Lambda(arg, body) ->
                    handle_Lambda_Apply(arg, arg_vals, body)
                Variable(name) ->
                    handle_Variable_Apply(name, arg_vals)
                el ->
                    add_error("Function application's first argument was not of type `lambda` but
                    get_Null_val()
        Let (arg, arg_val, body) ->
            val old_scope = pget_state()
            val retval = handle_Let(arg, arg_val, body)
            pset_state(old_scope)
            retval
        Letrec (arg, arg_val, body) ->
            val old_scope = pget_state()
            val retval = handle_Letrec(arg, arg_val, body)
            pset_state(old_scope)
            retval
        Equals(e1, e2) -> handle_Equals(e1, e2)
        EMult(e1, e2) -> handle_EMult(e1, e2)
        EAdd(e1, e2) -> handle_EAdd(e1, e2)
        ESub(e1, e2) -> handle_ESub(e1, e2)
        Null -> get_Null_val()
```

Figure 2: "General program analysis structure and abstract syntax token handler effect."

tools depending on the provided effect handlers. The generalized code would be mostly structural where the effect handlers functions would be invoked depending on the core language token encountered by this structural code as shown in Figure 2.

Ideally each handler could choose functions from a set of existing 'handler functions' which would handle one or more of the functions that the effect requires. Any remaining effect functions would then be coded by hand. This would reduce code duplication as, applications that use identical definitions of these functions could just reuse the same implementation (such as is the case for the handling of variables in the interpreter and type checker). Applications that need different, unique implementations for these effects however can just define them themselves (e.g. in a variable naming convention checker the code ran on variables will be different to that of a type checker).

This should greatly ease the process of creating code analyzers, as well as make the code produced less error prone as some already tested code is part of the solution. There is however the possibility that this would create lot of overhead and a lot of restrictions for the programmer. Whether this would be a good trade off depends on how much is gained from the generalization and code reuse, how costly the overhead and restrictions are, and what the goals of the project are.

4.2 The Created Code Base

The code base can be found on Github[2] *the repository also contains a* `README` *explaining what code each file contains.*

The final code base which was created consists of a few things, firstly an abstract data type had to be created which could represent Mini-ML programs. This Abstract Data Type (ADT) is very similar to the one shown in Figure 1 of Natural Semantics by G. Kahn [5], it has however been changed slightly.

The original abstract syntax did not explicitly allow for the modification of numbers, nor for any comparison between numbers. Thus an additional *addition*, *subtraction*, *multiplication* and *equality* primitives have been added to this abstract syntax. Furthermore it has been decided that recursive let bindings as well as lambda expressions should have their arguments type annotated. Without type annotation the exercise of type checking the values would have been far beyond the scope of this research, thus it was decided that type annotated arguments would be fine.

This Mini-ML `expression` type is important to write down programs in Mini-ML, but we also needed an `expression_type` type, both for the definition of lambda functions and recursive let bindings, as well as for a return type for the type checker. This `expression_type` consists of four actual values and an error value. It contains the `number` type, the `boolean` type, a type to denote `pairs` of types, and a function type, containing the argument type and the return type. Using the `pair` type of both `expression`s as well as `expression_type`s you are able to create any arbitrarily complex (but non-infinite) data types.

Besides the ADT there also needed to be some supporting functionality before the interpreter and type checker could be created, the `errors and warnings` effect and corresponding handler, and the `polymorphic_scope` effect and corresponding handlers. The `errors and warnings` effect was created because the already existing `exn` effect stops execution, our new effect just accumulates errors, to be requested once execution is stopped. The second effect of `polymorphic_scope` was more difficult to create. The scope should allow someone to add a polymorphic value to the scope, or add an unevaluated `expression` to the scope, this was accomplished by using a custom `closure` data type which can hold either of these two values. The Type definition of the polymorphic scope can be seen below, in Figure 3.

```
effect polymorphic_scope<a>
    fun pscope_contains(name: string) : bool
    fun pget(name: string) : closure<a>
    fun padd(name: string, value: a) : ()
    fun padd_with_closure(name: string, value: expression, clos: list<(string, closure<a>)>) : ()
    fun pget_state() : list<(string, closure<a>)>
    fun pset_state(newState: list<(string, closure<a>)>) : ()

type closure<a>
    Direct_value(value: a)
    Actual_closure(value: expression, closed_scope: list<(string, closure<a>)>)
```

Figure 3: "The type definition of the polymorphic scope effect."

During programming a problem was encountered where variables would escape their scope, in short, once a variable

[2]https://github.com/BrendanMesters/Koka-Linter

was declared it would never seize to exist. An elegant solution that was thought of would be to have a function of the scope effect which would instantiate a new instance of the scope effect with certain variables added. This, however, produced a lot of incomprehensible errors Because of this, it was eventually decided that a function would be added to request the current internal state, as well as a function to set the current internal state, this way the problems could be fixed in the code of the interpreter/type checker.

Lastly there where the actual implementations of the `type checker` and `interpreter`, both of which consist of a match case, matching on all possible type instances of the `expression` type. Next up different things happened when different values where encountered, if numbers or booleans where encountered their true value was returned in the case of the interpreter while the type checker returned their `expression_type`. Pairs where also simple, as this just called the function that is was a part of recursively on both its elements and then returned the pair of those two. In the interpreter, functions are returned exactly as they are and left for the apply handler code to be dealt with, while the type checker, on the other hand, immediately infers the return type and returns a `EFunc` function type.

```
Let (arg, arg_val, body) ->
    val old_scope = pget_state()
    args_and_vals(arg, arg_val, type_checker)
    val retval = body.type_checker
    pset_state(old_scope)
    retval
Letrec (arg, arg_type, arg_val, body) ->
    val old_scope = pget_state()
    args_and_types(arg, arg_type)
    val retval = body.type_checker

    // Make sure that the type annotation was actually correct.
    if arg_val.type_checker != arg_type
    then
        add_error("Typechecker: letrec had wrong type anotation")
        return ENull

    pset_state(old_scope)
    retval
```

```
Let         (arg, arg_val, body) ->
    val old_scope = pget_state()
    args_and_vals(arg, arg_val, test_interpreter)
    val retval = body.test_interpreter
    pset_state(old_scope)
    retval
Letrec      (arg, arg_type, arg_val, body) ->
    val old_scope = pget_state()
    args_and_vals(arg, arg_val, test_interpreter, True)
    val retval = body.test_interpreter
    pset_state(old_scope)
    retval
```

Figure 4: "Code duplication in `let` and `reclet` handling in the type checker and interpreter code respectively."

The code for handling the `Let` and the `Letrec` constructs are both very similar to each other, as well as near identical between the interpreter and the type checker, as can be seen in Figure 4. They each store the current scope, so that they can return to that one once they have handled their body, add the bound values to their respective argument names in a scope, and recursively call their own function on the body.

The recursive let binding does differ slightly between the type checker and the interpreter, as, with the type checker we add the assumed type to the scope and verify that the value indeed is of the given type. In the interpreter on the other hand we add the values to the scope, but we delay the interpretation of the values by adding a 4th element, namely `True` to the `args_and_vals` function call.

4.3 Problems encountered during the project

As mentioned in the introduction of Section 4, the original plan for the project was larger then what was eventually created. A lot of problems where encountered, when the complexity of the project grew there where increasingly more problems that arose. These problems arose, at least in part, due to the fact that Koka [3]:w is still a research language and because the scientific community is still trying to find the best way to work with effect handlers. A good example of a recent study in this field is the research by Zhixuan Yang on scoped effects, which might help with the reasoning about EHOP programs. [6]

The largest of the problem which was encountered during the programming of the more generalized solution can be seen in Figure 5. This was a simple mock setup of the effect handler to handle an interpreter in the generalized program structure shown in Figure 2. The effect typing is correct as `Num(n: int)`, `ETrue` and `EFalse` are all part of the expression type. Still this, seemingly correct, though trivial, code raises compiler errors. This shows that one large hurdle in the creation of programs that use effects in complex ways is the fact that the current EHOP languages are still in a somewhat experimental phase.

5 Experimental Setup

Testing code quality is usually done through frameworks which analyze your code and judge it on style, clarity and a myriad of other factors. These frameworks, however, are almost exclusively build on a language by language basis, this means that these tools are of little use when trying to analyze new languages such as Koka. Even for already existing languages its been shown to be difficult to correctly assess the clarity of code as shown by S. Scalabrino et al. [7].

For this reason most Program Language papers employ methods such as peer evaluation to assess different qualities about the code. Thus, this project planned on using a combination of substantiated reasoning about the code by the programmer and peer evaluation. Reasoning on the code will give an 'expert view' with substantiated argumentation and code examples where possible.

Note that 'expert view' is in quotations as it is actually a far try to call the programmer an expert in EHOP. But we used this word non the less as there are only very few people who have ever coded in EHOP and even less people who have worked on implementation code analysis tools in EHOP. Thus this will probably be the closest we get to 'an expert' in this specific field.

The peer review would provide insights into the general readability and understandability of EHOP code. Additionally it could have provide feedback on the concepts of EHOP from people who have not yet worked with those concepts,

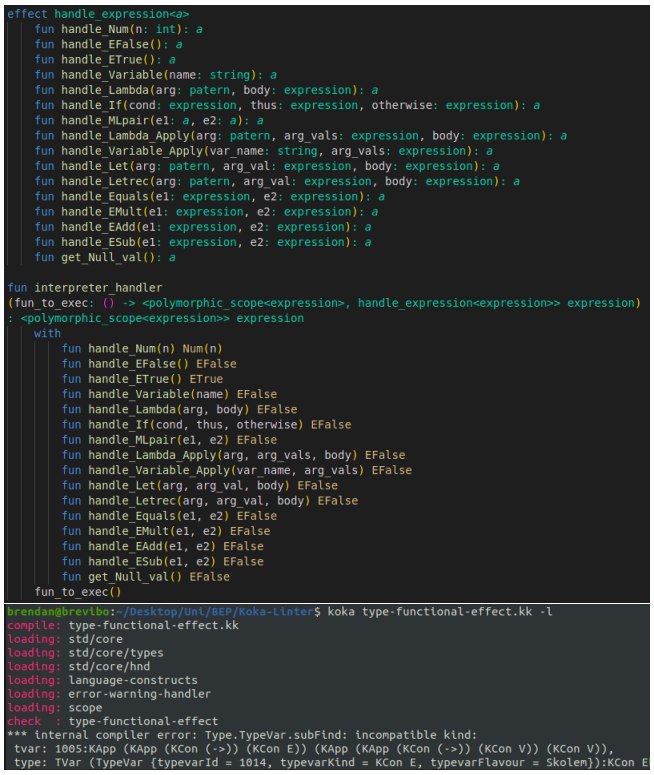

Figure 5: ”Mock handler for the generalized program analysis structure and the compiler error encountered when compiling it.”

which could have shown how natural the EHOP ideas come. However, the peer evaluation survey send out for this project did not receive as many answers as hoped, and due to the small sample size of the answers it has been decided that these will not be included in the paper. The answers that where received did seem to show some interesting commonalities, so this could be an interesting subject for future research and will be briefly touched on in Section 6.

6 Results

This research, just as most all Programming Language research papers, has opted to use somewhat subjective ways to answer the research questions as it would be infeasible to use objective tools as explained in Section 5. The research questions will each be answered by the programmer, these answers will be presented in a subsection, one for each sub-question. These answers aim to be as objective as possible, giving reasoning and examples for the statements made where ever possible.

Throughout this section there will also be a distinction between the current day practical answers to our questions and the more theoretical answers looking at future applicability. EHOP is currently still in its infant stages and some issues may be present now but not applicable when the languages and the concepts have matured a bit more.

In the following sub-sections I will try to answer the sub-questions of this paper by giving my own substantiated opinion. The nature of the questions often asks for either a subjective answer or a substantiated reasoning. Where possible I will try to keep the answers as objective as possible, but this will not always be possible.

It is also worth noting that there has also been a small peer review, using a survey to judge the code on a few metrics. The sample size of people who filled out the survey was to small to be included as a full result, but a consensus that seemed to arise from the few answers we got is that EHOP helps with the conciseness and functionality of the code This should, however, only be taken as a suggestion to this being the case as opposed to an actual result, due to the small sample size of 4 answers.

RQ1: Do the concepts present in EHOP translate well to program analysis tools?

Some of the ideas and advantages of EHOP translate quite well into the domain of program analysis tools, while others are not as applicable. The area where program analysis tools do greatly benefit from the concepts present in EHOP is when it comes to code generalization. EHOP should ultimately allow you to generalize your code to a degree similar to what was shown in Figure 2. Allowing effect handlers to use the same, slightly generalized, implementation of similar pieces of code such as what Figure 1 shows. This ”hyper generalization” however, does still have some issues in the current programming environment, for more detail see Sub-Section 4.3. These issues are partly due to the fact that Koka is still a research language that is in development. I think it very likely that these issues will be either solved, or way more manageable once the EHOP paradigm has matured more.

Another big benefit from EHOP languages is that you can modify the behaviour of large sets of code, spanning many files and functions with great ease. Code analysis tools do not benefit from this feature, as most code analysis tools are relatively small in scale. This benefit is mostly enjoyed by large programs that want to be able to use priorities to change what they can do and how they can do it.

RQ2: Does EHOP allow for more code reuse then otherwise possible?

The answer to this question in theory should be a resounding yes, in practice however this has proven to be more difficult. There are many parts of the code that are very similar if not near identical in many different code analyzers, look at Figures 1,2,4 for example, both structural code as well as specific code snippets often recur. It should be possible to set up a code base where this code duplication can be eliminated, at least for the most part. The only issue with this idea is that, currently, the state of Koka does not seem to allow for these pieces of code to be created, as compiler errors are thrown when polymorphic effect handlers are used in complex ways as can be seen in Figure 5 (for more detail see Section 4.3).

RQ3: How does EHOP programming effect the readability of the code?

The improvements of readability that EHOP provides really depend on what it is compared to. Comparing an EHOP implementation of a code analysis tool with one in a functional

language will have the EHOP implementation be much more readable. A functional language would have to explicitly pass variables such as the current scope and the accumulated errors and warnings, both up through recursive function calls, as well as down through return values. This would create a lot of visual clutter that would negatively effect the clarity of the code. In EHOP these functionalities that the code needs are mentioned once at the type definition of the function and can, thereafter, just be used and passed both into function calls, as well as out of them implicitly.

Comparing EHOP to an OOP language makes the comparison a bit more nuanced however. For small programs where the code remains in one file OOP will be more readable, as the functions that would be wrapped in effects in EHOP can just reside in the global scope. If the codebase is larger however you either run into the problem that the one file you use becomes very large, thus making the code less clear. The other option would be to spread the code across multiple files, but then you would have to explicitly pass many extra values trough function calls, which will have a negative effect on the readability of the code.

RQ4: How quickly can an EHOP language be picked up by someone with prior functional programming experience?

Simple programs are quite trivial, as they are very similar to functional programming. Actually working effectively with effects takes some time however, and I feel like even 3 months is not enough to be able to use effects to their fullest extend. That being said, grasping the concepts well enough be be able to program with effects well enough for most tasks took roughly 2 weeks. And after 4 weeks I was comfortable with what effects could and could not do in most situations. If you really want to use effects to the fullest however I believe that you will need multiple months of experience programming with effects, just as with any other language that you want to excel at.

7 Responsible Research

This paper will not be perfectly reproducible due to the nature of its research. This paper looks at a single programmers attempt to learn EHOP and create a program with it that is to be analyzed. Since every programmer will handle this task slightly differently we can assume to see slightly differing results if the research is reproduced. We do not believe that there are any ethical issues which came up during the project as this paper is just a practical look at a new programming concept.

8 Discussion

Projects never run exactly to plan as we all know, and so this project too had grander plans in the beginning. The original plan was to generalize both the interpreter as well as the type checker so that they would not have to have any duplicate code between them. This would've been done by having a polymorphic function, which would traverse the abstract data tree, calling functions of specific effect handlers to execute the parts of the code which differ between different code analysis tools. You would then have separate effect handlers for interpreting or type checking, and depending on which one you'd provide to this polymorphic 'skeleton code' you could get vastly different outcomes.

However, this did not go as smoothly as hoped, and compiler errors made this almost impossible to implement as shown in Sub-Section 4.3. These compiler errors left the programmer in the dark about what they had done wrong, and the error messages where becoming increasingly less intelligible as the code used multiple layers of polymorphic abstractions. These problems where rather large and halted any further progress as it was not clear what went wrong or how it could be fixed.

Thus it was eventually decided that it would be best to decrease the scope of the research slightly to make sure that deadlines could be made. This is an example of effect polymorphism breaking on more complex and sophisticated use cases as noted by J. I. Brachthäuser et al. [8]. This also does not seem to be an issue with the EHOP programming paradigm, but rather a consequence of the fact that Koka [3] is still a relatively young language. The documentation does not yet cover everything that Koka has to offer and the compiler still has some small bugs in it.

9 Conclusions

This research has shown that EHOP does have many different benefits when it comes to writing program analysis tools, however, it has also shown that the current research languages might not yet be ready for such complex programs.

The concept of interchangeable functionality which is inherent to EHOP is very well represented in static code analysis tools, as, across many different tools, the structural code is often the same and even the code to handle specific language tokens is often similar between different analyzers.

This allows for more effective code reuse, as implementations which would not ordinarily be able to be shared across different code analyzers are now able to be shared if the code structure is set up well. However, these improvements in code reuse are not currently feasible due to the shortcomings that Koka [3] still has as it is still a research language.

The readability also either improves or stays as good as when Functional Programming of Object Oriented Programming would've been used, depending on the size of the program. And EHOP is relatively easy to pick up if someone already knows functional programming. Mastering EHOP does take a good amount of time however, as the concepts of effect handlers do become somewhat vague when they are used in more complex situations.

10 Future Work

There are some areas which could still greatly benefit from additional research. The first of these areas would be further research into higher levels of abstraction and thus more code reduction should be very useful, as it could give more details into the issues and possible solutions which occur when such highly generalized code is written.

Another interesting research topic would be to see if the findings of this paper extend to other static code analyzers. This paper only analyzed a type checker an interpreter, but the applicability to say a code analyzer that checks that variable naming conventions are upheld might also be interesting. This would most likely require a different programming language to be analyzed though, as Mini-ML is to simple to have complex variable naming rules.

Furthermore research into the application of EHOP in other sets of programs should be considered, as this could give more general statements about the EHOP programming paradigm, as well as shed light on the possible real world applications of EHOP. Research into methods to judge EHOP code could also be interesting, as the different paradigm will most likely have different ways of measuring code quality.

References

[1] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," *Programming Languages and Systems*, pp. 80–94, 2009.

[2] D. Leijen, "Type directed compilation of row-typed algebraic effects," *ACM SIGPLAN Notices*, vol. 52, pp. 486–499, 05 2017.

[3] D. Leijen, "The koka programming language." https://koka-lang.github.io/koka/doc/book.html.

[4] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do," *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 01 2017.

[5] G. Kahn, "Natural semantics," *STACS 87*, pp. 22–39.

[6] Z. Yang, M. Paviotti, N. Wu, B. van den Berg, and T. Schrijvers, "Structured handling of scoped effects," in *European Symposium on Programming*, pp. 462–491, Springer, Cham, 2022.

[7] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 595–613, 2021.

[8] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effects as capabilities: Effect handlers and lightweight effect polymorphism," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.