# The effect of EHOP on the writing of Program Analyzers

Author: Brendan Mesters
Supervisors: Cas van der Rest
Responsible Professor: Casper Bach Poulsen

## 1. Background

### Effect Handler Oriented Programming (EHOP)

- Seperation of concerns for side effects.
- Isolates application logic from side effect handeling.
- Effects: Interface denoting an effects capabilities.
- Effect handlers: Concrete implementation of effects functionality.
- Effectful functions: Function that requires certain effects.

Figure 1 shows an effect, effectfull function and the handeler for the effect.

EHOP excells at creating customizable functions since the effect handlers can be changed to change the functionality.

```
// A generator effect with one operation
effect yield<a>
  fun yield( x : a ) : ()

// Traverse a list and yield the elements
fun traverse( xs : list<a> ) : yield<a> ()
  match xs
    Cons(x,xx) -> { yield(x); traverse(xx) }
    Nil        -> ()

fun main() : console ()
  with fun yield(i : int)
    println("yielded " ++ i.show)
  [1,2,3].traverse
```

Figure 1: Example of effectfull code, implementing a yield effect to print the given value to console [1]

### Static code analysis

- Analyzes source code for errors and warnings.
- Traverses the abstract syntax tree of a program.
- There exist many different static code analyzers.
- Different analyzers often have similar or identical pieces of code.

## 2. Research Question

**How does the EHOP programming paradigm effect program Analysis Tools.**

- Do the concepts present in EHOP translate well into Program analysis tools?
- Does EHOP allow for more code reuse then otherwise possible?
- How does the EHOP programming paradigm effect the readability of the code?
- How quickly can EHOP be picked up by someone with prior functional programming experience?

## 3. Methodology and process

During the project a **type checker** and an **interpreter** have been made in the **EHOP language Koka** [1] for the toy language **Mini-ML** [2].
The written code also included data types to **represent Mini-ML code**, as well as an effect to handle accumilative **errors and warnings**, and an effect to keep track of the **scope** during code analysis.
Various **Mini-ML programs** have been written as values within the main.kk file.
The research questions were answered by **substantiated reasoning**, using **examples** from the source code where ever possible.

## 4. Results

Do the concepts present in EHOP translate well into program analysis tools?

**Yes**,
- Effect Handlers allow for high levels of code generalization
- Functionality can be shared between code analyzers. See Figure 2 for an example on identical code.
- Code analyzers use the same 'code skeleton' consisting of a large match case to traverse the abstract data tree of the program.

### Does EHOP allow for more code reuse then otherwise possible?

The answer to this sub-question is twofold.
- In Theory: Yes, EHOP can eliminate vast parts of duplicate code and allow for code sharing between different code analysis tools.
- In Practice: Koka does not seem ready for code which uses effect handlers on a highly abstracted level.
  This issue is, however, almost certainly a result of the fact that Koka is still a research language.

### How does the EHOP programming paradigm effect the readability of the code?

**Positively**,
EHOP allows for functionality to be passed trough function calls implicitly, this makes the function calls less messy as they use less arguments.

```
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 2: The near identical Variable handler code for the type checker and the interpreter

### How quickly can EHOP be picked up by someone with prior functional programming experience?

**It depends,**
- Simple programs are very similar to Functional programming and easy to write.
- It takes roughly 2 weeks to get familiar with EHOP.
- It takes roughly 4 weeks to understand effects and structure code around it.
- It will take multiple months to 'master' EHOP

## 5. Conclusion

- Static Code Analyzers do benefit from EHOP as it can help to reduce duplicate code and visual clutter.
- The current EHOP languages are still quite young and are not yet ready to be used in production code.
- EHOP is a prommising new programming paradigm that can easily be picked up by anyone with functional programming experience.

## 6. Future Work

- Peer research into the clearity of EHOP code, and the understandability for non-ehop programmers.
- Research into the applicability of EHOP for other programs
- Extending this research to more different kinds of code analyzers.

## 7. References

1. https://koka-lang.github.io/koka/doc/index.html
2. G. Kahn, “Natural semantics,” STACS 87, pp. 22–39.