Improving the quality of code in IO intensive applications through effect handlers

Author: Sam Streef (S.L.Streef@student.tudelft.nl) **Supervisors**: Jaro Reinders & Cas van der Rest **Professor**: Casper Poulsen

1. Background

Effect Handler Oriented Programming (***EHOP***) is a programming paradigm that provides a separation of concerns by abstracting code into effects and effect handlers.

An **effect** is a definition of operations that can be used in a function that is using this effect.

An ***effect handler*** is an implementation of the operations of an effect.

Multiple effects can be used in a function and the effect handlers that run an effect determine its implementation and functionality.

6. Conclusion

For IO intensive applications such as an HTTP server:

- EHOP can improve readability, maintainability and modularity.
- EHOP adds a performance overhead in both **memory** and **runtime**.

However, since IO intensive applications are performance driven, the improvement in quality of code that EHOP provides does not outweigh the overhead that it adds.

7. Future work

Extensions to this work could be made by:

- Running the same experiments in a language with effects and effect handlers built-in
- Running the experiments for a different application such as a serial communication application.
- Analyzing the experiments with more concise metrics such as cyclomatic complexity, coupling and cohesion.

```haskell
-- Effect
data Logging m a where
    Log     :: String -> Logging m ()

-- Polysemy function generating effect operations
makeSem ''Logging

-- Effect handlers
type LoggingHandler = Sem '[Logging, Embed IO] ()
                   -> Sem '[Embed IO] ()

runConsoleLogging :: LoggingHandler
runConsoleLogging = interpret $ embed . \case
    Log s   -> putStr s

runFileLogging :: LoggingHandler
runFileLogging = interpret $ embed . \case
    Log s   -> appendFile "log.txt" s
```

1 *Definition of a Logging effect and two effect handlers in Haskell using the Polysemy library.*

```haskell
program :: Sem '[Logging, Embed IO] ()
program = do
    log "Hello "
    log "World!"

main :: IO ()
main = program          -- [Logging, Embed IO]
    & runConsoleLogging -- [Embed IO]
    & runM              -- []

main' :: IO ()
main' = program         -- [Logging, Embed IO]
    & runFileLogging    -- [Embed IO]
    & runM              -- []
```

2 *Using the Logging effect with different implementations.*

```haskell
f :: Sem '[Logging, State (Maybe String)] ()
f = do
    Logging.log (...)
    (maybeString :: Maybe String) <- State.get
    ...
```

5. Qualitative results

Readability

+ Effect signatures and effect handlers show the functionality of a function.
+ Monads such as state can be used and combined with other monads in a single do notation, reducing lines of code.

Maintainability

+ Using effects allows for changing effect behavior dynamically by using different effect handlers. Functions with effects can be tested with simple handlers whilst application code uses a different handler.
+ Effects can have operations added or changed whilst keeping the application type correct.

Modularity

+ Code can be written type correctly by only having an effect available and without an effect handler.
+ Effects are backwards compatible if functionality is added.

All these results are based on the effects for buffering, logging, file reading and request handling.

2. Research Questions

How does EHOP affect modularity, readability and maintainability of code for IO intensive applications when compared to implementations not using EHOP?

- Does EHOP improve the *readability* of code?
- Does EHOP improve the *maintainability* of code?
- Does EHOP improve the *modularity* of code?
- Does EHOP improve the *response time* of IO intensive applications?
- Does EHOP increase the *memory usage* of IO intensive applications?

3. Method

Execution of the experiments

Building a basic HTTP server from a TCP socket in Haskell whilst;

- Using effect handlers
- Not using effect handlers

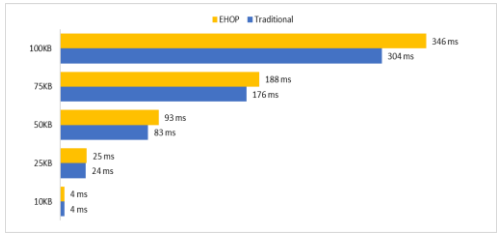
(The *Polysemy* library is used for effect handlers)

Evaluation of the experiments:

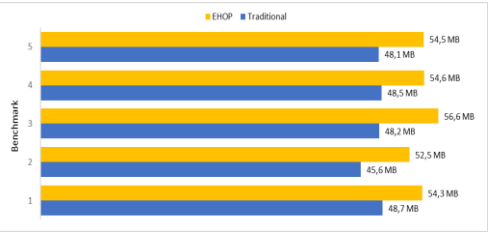
- Compare the readability, maintainability and modularity of the programs
- Measure the response time of each program
- Measure the memory usage of each program

Performance benchmarks run POST requests with a payload to an endpoint on the HTTP server that sends the payload in reverse back.

4. Quantitative results



1 *The results show that EHOP has a **response time** overhead compared to the traditional paradigm.*



2 *The results show that EHOP has a **memory usage** overhead compared to the traditional paradigm.*