Effect Handler Oriented Programming for Data Processing Applications

ALI BASARAN
Supervisor(s): JARO REINDERS, CASPER 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 or EHOP for short, is a new programming paradigm aiming to achieve separation of concerns in code which will lead to modular, readable and maintainable code. Since EHOP is significantly new, it is important to assess and compare it against traditional, commonly used paradigms in order to see if a wider adoption of EHOP would prove beneficial to computer science. In this research, EHOP was compared with traditional paradigms under the context of data processing applications. An Excel-like command line application called "MiniExcel" was implemented from scratch. Moreover, "Hierarchical EHOP", a new structural pattern for EHOP was defined which enforces rules between concepts and produces a readable code structure. The main conclusions of this research can be summarized by the following statements. EHOP produces more modular, readable and maintainable code compared to traditional paradigms. Implementing additional concepts and updates to code is seamless using EHOP, yet the lack of development in EHOP's ecosystem raises frustrating errors and requires the developer to implement libraries that are usually built-in for languages that support traditional paradigms. Functional programming produces faster running code, but EHOP is more memory efficient. Therefore, for applications that interact with users EHOP is the better choice and for applications that only execute code functional programming is more suitable.

1 Introduction

In the ever evolving study of computer science, research on new programming paradigms is bound to happen. One of the newest results of such research is the effect handler oriented programming (EHOP) paradigm. EHOP aims to achieve separation of concerns in software which produces readable, modular and maintainable code. In EHOP, programmers define operations that represent side-effects, and implement these operations under effect handlers that are separate from the main application in which the operations are called. An example code snippet written using EHOP is shown below:

```
// Define emit operation as effect
effect fun emit(msg : string) : ()

// Call emit in application logic
fun hello()
  emit("hello world!")

// Handle emit in separate handler logic
pub fun handle-emit()
  with handler
    fun emit(msg) println(msg)
  hello()
```

The code snippet displays typical utilization of EHOP. Firstly, a "emit" operation is defined as an effect. Then the application code ("hello") calls the emit operation, raising an emit effect. Until now, no implementation of emit is given. Finally in a separate section, a "handle-emit" function is defined which works as a handler for the application code. In this example, the emit effect is handled to print its input to the console. At the end of the handle-emit function, the application code is called to handle the application.

The goal of this research is to study and assess EHOP against "traditional" paradigms by implementing a data processing application using EHOP. "Traditional" paradigms refer to the commonly practiced programming paradigms such as functional programming and object oriented programming. Since EHOP is a newly proposed paradigm, it is important to compare it against traditional paradigms to display its usefulness and analyze if a wider usage of EHOP will prove beneficial in software engineering.

The reasoning behind implementing a data processing application rather than other applications is that data processing applications are commonly used by individuals and companies thus an assessment of EHOP on them is more impactful. In this research, the implemented data processing application is called MiniExcel, which is a miniature, command-line version of Microsoft Excel [1] that implements a small set of Excel's functions [2]. How one can interact with MiniExcel and how MiniExcel was implemented is further described in Section 3.2.

Extensive work exists on EHOP, analyzing the expressiveness and the efficiency of the paradigm, most notably by Hillerström [3]. Hillerström implemented parts of the Unix operating system [4] using EHOP, displaying the versatility and usefulness of EHOP in terms of code abstraction. Moreover, they proved that for some programming problems, using EHOP provides asymptotically faster implementations compared to not using it.

However, even the extensive research of Hillerström leaves some questions unanswered due to EHOP's recency. From a qualitative perspective, Hillerström has shown that EHOP can achieve code abstraction by implementing some of Unix's functionality. However, EHOP's efficiency in producing readable code is still unanswered in the context of commonly used, traditional applications such as games or data processing applications. Moreover, Hillerström assesses EHOP in concrete aspects and does not observe the developer experience when implementing applications using EHOP. Having an idea on what to expect is important for developers that want to use EHOP in the future. From a quantitative perspective, Hillerström has shown that using EHOP can yield asymptotically faster implementations on some programming problems, yet it is unknown if that is the case for the run-time and memory characteristics of mainstream programs such as games and data processing applications. Aforementioned unanswered aspects are important to answer and compare to traditional paradigms, as they will contribute in the assessment of the usefulness of EHOP in software engineering.

The main research question to answer is "How does using EHOP for implementing a data processing application affect the **modularity**, **readability** and **maintainability** of code compared to traditional paradigms?". This question is split into four sub-questions where EHOP is compared to traditional paradigms in qualitative and quantitative aspects:

1. How does using EHOP for implementing an application affect the **readability of code** compared to using traditional paradigms?
2. How does the **experience of developing and maintaining an application** using EHOP compare to traditional paradigms?
3. How does the **runtime** of an application implemented using EHOP compare to the **runtime** of similar applications implemented using traditional paradigms?
4. How does the **memory characteristics** of an application implemented using EHOP compare to the **memory characteristics** of similar applications implemented using traditional paradigms?

The main contributions of this paper are:

- MiniExcel, an open-source Excel-like application implemented using the EHOP paradigm;
- An explanation of Hierarchical EHOP in Section 3.2.2, which defines a readable code structure that enforces rules between concepts;
- Examples on how EHOP can be advantageous when implementing certain concepts in Section 3.2;
- A comparison of EHOP with traditional paradigms over various aspects in Section 4.

The structure of this report is as follows. In Section 2, the methodology of the research will be explained. In Section 3, how MiniExcel works and how MiniExcel was implemented will be described. In Section 4, the setup and the results of the experiments will be displayed. In Section 5, the ethical aspects and the reproducibility of this research will be analyzed. In Section 6, the results obtained in Section 4 will be discussed further. Finally in Section 7, the research question will be answered and possible future work will be examined.

2 Methodology

In this section the approach taken to answer the research question will be explained. The methodology of this research is separated into three steps. The first step is to implement the data processing application, MiniExcel. The second step is to analyze MiniExcel on various metrics such as readability. The third and final step is to compare EHOP with traditional paradigms using the analysis results.

2.1 Implementation

The data processing application implemented for this research is called MiniExcel and as the name suggests, is similar to Microsoft's Excel [1]. The motivation behind deciding to implement an Excel-like application is that compared to other data processing applications, Excel introduces concepts in which EHOP can be utilized in clever ways such as handling dependencies between spreadsheet cells. The key differences between MiniExcel and Excel is that MiniExcel is interacted through the command line interface and MiniExcel only implements Excel's summation, subtraction, division and multiplication functions [2].

A basic set of four functions is sufficient to answer the research question because the goal of this research is to assess EHOP and the usage or assessment of EHOP would not change if four or fifty functions of Excel were implemented. To elaborate, in MiniExcel, user commands are first received as strings such as "$A1 = \mathrm{SUM}(A2, A3)$" and the conversion from string to evaluate-able code is implemented through helper functions which do not use EHOP. Therefore, implementing more functions would only require changes on helper functions which do not use EHOP.

In their article on build systems "Build Systems a la Carte", Mokhov et al. [5] state that "Excel is a build system in disguise". Throughout implementation, MiniExcel closely follows their article and deep down works as a build system with a spreadsheet interface. The reason behind choosing Mokhov et al.'s article is the fact that their article is hosted online by Microsoft, the company that created Excel, thus contains the most accurate information on the intricacies of Excel.

On a lower level, the application is implemented in Koka [6; 7]. There are a few reasons for selecting Koka amongst other candidates, most notably Haskell [8] and Frank [9]. Firstly, Koka has built-in support for EHOP, which from the choice set, eliminates languages that depend on third party libraries to support EHOP such as Haskell. From the languages that have built-in support for EHOP, Koka is the language that is the most actively maintained and improved. Therefore, Koka was selected as the language of the application.

2.2 Analysis

In this research, analysis is done on four aspects: readability, development and maintenance experience, runtime characteristics, and memory characteristics. Each analyzed aspect refers to one sub-question stated in Section 1 thus is essential for answering the research question. Below are the approaches taken to analyze each aspect.

Readability is measured both objectively and subjectively in this research. Objectively measuring readability is difficult as it is a subjective criteria. However, there are generalizable aspects of code that allow for an accurate objective measurement of code readability. The objective model introduced by Posnett et al. [10] will be used because their model is an improved version of Buse and Weimer's readability model [11]. Scalabrino et al. [12] also discuss ideas that improve the accuracy of both models but while doing so complicates them. Since objective models are used as trends rather than facts, given the time frame of this research, the ideal choice is Posnett et al.'s model which provides more than sufficient accuracy in measuring readability and is simpler than Scalabrino et al.'s model.

Posnett et al.'s model revolves around three metrics: lines of code, entropy and Halstead's volume. Posnett et al. describe these metrics as follows. Entropy measures the com-

plexity of a document X and is calculated by:

$$Entropy(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

where

$$p(x_i) = \frac{\text{count}(x_i)}{\sum_{j=1}^{n} \text{count}(x_j)}$$

and the function count is the number of occurrences of token x_i in document X. Halstead's volume (HV) depends on two other metrics, program vocabulary and program length. Program vocabulary is the count of unique operators and operands whereas program length is the total number of operators and operands used in document X. Measurements of these metrics are fitted into the readability model of Posnett et al. :

$$\frac{1}{1 + e^{-z}}$$

where

$$z = 8.87 - 0.033HV + 0.40 \text{ Lines } - 1.5 \text{ Entropy}$$

After fitting the metrics into the Posnett et al.'s model an accurate value for an application's readability is achieved. Subjectively, the aspects of the application's programming paradigm which produces readable code is discussed. Development and maintenance experience is subjectively measured by the researcher by taking notes throughout developing and maintaining the application. To analyze runtime and memory characteristics, a predetermined list of commands are executed in the application. While executing commands, built-in time and memory analysis tools of the programming language are utilized to measure runtime and memory characteristics.

2.3 Comparison of Results

This section explains the methodology used to compare the analysis results of the application against traditional paradigms. After finishing this step, the sub-questions and the research question could be answered.

To complete this step, a similar application implemented with a traditional paradigm is required. The selected "similar application" is implemented by Mokhov et al. [13], which is the source code of their detailed article [5] on build systems. Their application is suitable for comparison with MiniExcel because their application implements the concepts of MiniExcel using the functional programming paradigm. Their code contains implementations of multiple build systems including Excel. Therefore from their codebase, relevant code to run the Excel system is extracted into a single script to make comparisons with MiniExcel easier. Then to obtain comparable results, relevant code will go through the analysis process defined in Section 2.2 with the exception of development experience.

By using Mokhov et al.'s application [13], comparisons on readability, runtime and memory are easily achieved; the same cannot be said for development experience. Since Mokhov et al.'s application is not implemented by the researcher it is impossible to make comparisons on development experience. To solve this problem, the researcher compared their previous experience using traditional paradigms with their recent experience using EHOP.

3 MiniExcel

This section first displays MiniExcel's functionality, then dives into the implementation of concepts that are crucial in MiniExcel with a focus on the interesting use cases of EHOP.

3.1 Interacting with MiniExcel

MiniExcel, similar to Microsoft's Excel [1], is also a spreadsheet consisting of cells. However, MiniExcel is interacted by entering commands into the command line and after each command the updated spreadsheet is shown in the interface. Each cell is referred by an unique key that is a combination of any number of letters followed by any number of digits. For example the key A1 refers to the cell that is in column A and row 1. Cells can contain three types of "tasks": decimals, keys of other cells and functions. By evaluating a task MiniExcel retrieves the value of the cell as a decimal. MiniExcel implements four functions SUM, MINUS, MULT and DIV which sums, subtracts, multiplies and divides their parameters respectively. To assign a value to a cell, the user has to enter a command of the following format: "cell-key $=$ task". To further explain how MiniExcel works an example interaction with the application is displayed below:

$$A1 = 1.1$$
$$A2 = A1$$
$$B1 = \text{SUM}(A1, A2)$$
$$B2 = \text{SUM}(B1, 2)$$

In MiniExcel after each assignment the updated spreadsheet is printed. However, for the sake of saving space and avoiding repetition, after entering the last command the following spreadsheet would have been printed out.

	A	B
1	1.1	2.2
2	1.1	4.2

As evident from the spreadsheet above, MiniExcel only displays cells as decimals. What if the user wants to see a description of the task inside the cell? The command that fulfills this need is the "desc cell-key" command, which retrieves the function that is used to evaluate the cell of a given key. For example, in a spreadsheet where "$A1 = \text{SUM}(A2, A3)$", calling "desc $A1$" would return "$\text{SUM}(A2, A3)$".

The spreadsheet in MiniExcel is technically infinite. Since infinite cells cannot be printed to console, the spreadsheet is displayed as if the user is looking at it through a window. It is up to the user to determine the size of the window as well as where it points to. To change the size of the window the command "window h w" is used where h is the height value and w is the width value. For example "window 10 10" would set a 10 by 10 window which is the default in MiniExcel. The window's pointer is also crucial when going through a spreadsheet. The pointer is moved to a different cell with the "mv-to cell-key" command. The example below shows a 2 by 4 window with a pointer on C1.

	C	D	E	F
1	–	–	–	–
2	–	–	–	–

3.2 Implementing MiniExcel

In this section, first the concepts that create the core of MiniExcel and their implementations will be explained. Then, the term "Hierarchical EHOP" will be discussed. Then the concepts that optimize the speed of MiniExcel and their implementations will be shown.

3.2.1 Core of MiniExcel

The core of MiniExcel starts with the store which is used to store and retrieve cells. Cells contain a task and a string that describes the task. Then, task evaluation logic is needed to "fetch" the value of a cell. Fetching, converts the task of the cell into a displayable number. Finally, logic that converts a spreadsheet into a string is needed because in MiniExcel the spreadsheet is shown to the user after each command entry.

Each concept has their own effect definitions and underlying operations; store, task and visuals respectively. In MiniExcel, store defines three operations: set-key which sets the value of a given key to a given cell, get which retrieves a cell given a key and get-nonempty-keys which returns all keys that contain a cell.

Task only defines a single operation fetch which, given a key, converts the task of the cell into a decimal. The main fetch operation acts as a redirect to the helper functions. The helper function that does actual fetching is fetch-task-helper which works as follows. There are three types of cell tasks: decimals, key of other cells and functions. Decimal kind is directly fetched into a decimal. Key kind is fetched by calling fetch on the key recursively. As one might notice, this helper function raises the fetch effect by itself which will be important later. Finally the function kind is fetched by recursively calling the helper function on the function's parameters. This is because functions can have other functions inside (eg. $A1 = \text{SUM}(A2, \text{MULT}(A3, A4))$).

Finally, visuals defines print-spreadsheet which returns the spreadsheet as a string. This operation fetches all keys inside the application window one-by-one. The order in which the keys are fetched is called a "calc chain" and in MiniExcel it starts from the windows pointer and ends at the farthest key from the pointer. For example, for a 2 by 2 window with a pointer on key "A1" the calc chain would be $[A1, A2, B1, B2]$. The operation print-cell-desc given a cell key returns the description of the requested cell's task. Finally, update-view-window and update-pointer, change the size of the window and the pointer of the window respectively. However, defining these operations under effect umbrellas is not enough to implement a working application.

Each effect has its own handler with the format of "effect-name-handler". Moreover, Koka allows handlers to be defined as functions. In MiniExcel, first a concept is defined as an effect with operations underneath it. Then, below it is a handler function that handles the effect. For example, the pseudocode of the store effect and store-handler is shown below:

```
effect store
  fun set-key(k : string, v : cell) : ()
  fun get(k : string) : cell
  fun get-nonempty-keys() : list<string>

fun store-handler(action: () -> <store|_e> a): _e a
  ...
  with handler
    fun set-key(k, v)
        // implementation ...
    fun get(k)
        // implementation ...
    fun get-nonempty-keys()
        // implementation ...
  action()
```

As evident from the pseudocode, a store effect and a store-handler is defined that implements the operations of the store effect. The function store-handler can be interpreted as a handler that takes a function action as input which returns a value of type a and raises a store effect as well as other irrelevant effects. Then, the function handles the store effect raised by action and raises the rest of the un-handled effects $\_e$. In code this is done by first defining a handler and then calling action below. This allows the handler to catch and handle effects raised by action.

3.2.2 Hierarchical EHOP

All of the handlers of the explained concepts in Section 3.2.1 are connected to each other under a linear hierarchy, which in this research is referred to as "Hierarchical EHOP". The idea behind Hierarchical EHOP is that a member of the hierarchy can call operations of other members under it but not on top of it. This design allows for an easier grasp on the application structure and imposes a rule on how handlers work with each other. By using EHOP such hierarchy can be achieved as handlers can be combined.

In MiniExcel, all of the aforementioned handlers are combined under a single handler handle-app as follows:

```
fun handle-app(action : () -> <app-effects|_e> a):
↪  _e a
  with store-handler()
  with task-handler()
  with dirty-bit-handler()
  with build-handler()
  with visuals-handler()
  action()
```

where app-effects are the effects of the concepts of MiniExcel such as store, task and visuals. The combination of the handlers creates a linear hierarchy where visuals-handler is on the top and store-handler is on the bottom.

3.2.3 Optimizing MiniExcel

One might notice that there are two other handlers in the code snippet in Section 3.2.2 that are not explained yet: build and dirty-bit. This is because MiniExcel can work without the two. However, they are crucial concepts in Excel that optimize the spreadsheet building speed. Before diving into the two, it is important to understand why they are so crucial. Take the following set of commands as an example:

$$A1 = \text{SUM}(B1, B2, ..., B1000)$$

$$A2 = \text{SUM}(A1, A1)$$

Let the calc chain be $[A1, A2]$. When this calc chain is executed, first a fetch on A1 will be called. Since A1 is the sum of all cells between B1 and B1000, assuming the tasks of B1

- B1000 are decimals, an extra 1000 fetches will be called, making a total of 1001 fetches. Next on the calc chain is A2, which is in total three fetch calls one for itself and two times for A1. Since we know A1 is 1001 fetch calls, fetching A2 requires $1 + 1001 + 1001 = 2003$ fetch calls. In total to execute the whole calc chain, fetch is called 3004 times which begs the question: "Are all of these calls necessary?" The answer is no and this is where the concepts dirty bit rebuilder and restarting scheduler come in.

The dirty bit rebuilder aims to only fetch cells that are "dirty". A dirty cell is a newly entered or changed cell and a clean cell is a cell that was not affected after change in the spreadsheet [5]. This logic is implemented under the effect dirty-bit which defines three operations: is-dirty which given a key returns true if the cell that the key points to is dirty, clean which cleans the cell of a given key and put which given a key and a value, puts a cell with value into key in the spreadsheet. The description of put seems exactly the same as the set-key operation of store. However put, within the dirty-bit-handler sets the newly entered key as well as all other keys that depend on the entered key as dirty. Then after making keys dirty, the rebuilder delegates a set-key call to the store-handler which enters the cell into the spreadsheet. Hierarchical EHOP allows for the dirty-bit-handler to interact with the store-handler as the dirty-bit-handler is higher in the hierarchy.

Using EHOP, dependencies of a cell can be calculated in a clever way. Recall that the helper function fetch-task-helper when fetching a cell key calls fetch internally and raises the fetch effect. To calculate dependencies, when fetch-task-helper calls fetch internally, the raised fetch effect is handled to extract the key into a dependency list. In MiniExcel this is done as follows:

```
pub fun calculate-dependencies(t : task): <exn,
↪  div|_e> list<string>
  var deps := []
  with handler
    fun fetch(k)
      deps := deps ++ [k]
      // return any decimal
      decimal(1.0)
  fetch-task-helper(t)
  deps
```

With the knowledge on the dirty bit rebuilder, the restarting scheduler can be explained. Mokhov et al. [5], describe the restarting scheduler as follows. Given a calc chain, it starts to evaluate keys from left to right. In evaluation, the scheduler first checks the dependency of the current key, if one of its dependencies is dirty then the scheduler stops evaluating the current key and starts evaluating the dirty dependency, "restarting" the evaluation. Only after the dependency is evaluated it resumes with the evaluation of the initial key.

In MiniExcel, the restarting scheduler is implemented with the build effect that defines a single operation eval which, given a cell key, evaluates the cell's task and returns a decimal. In eval, first the key is checked to see if its cell is dirty. If it is clean then eval returns the last evaluated value which is stored in a hashtable initialized in the build-handler. If the key is dirty, eval calls fetch-task-helper and, similar to the dependency calculation, handles fetch effects internally. By doing so, whenever fetch is raised eval checks if the fetched key is dirty, if not it retrieves its value from the hashtable, if so another fetch is raised to be handled by the task-handler. This flexibility is once again allowed by EHOP as handlers can handle the same effect differently. Below is a pseudo code of the build-handler.

```
fun build-handler(action : () -> <build|_e> a) : _e a
  var value-map : hashtable<decimal> := hashtable()
  with handler
    fun eval(key)
      if !is-dirty(key)
        retrieve key from value map
      else
        with handler
          fun fetch(k)
            if !is-dirty(k) then
              return k's value from value map
            else
              make fetch call to the task-handler
              clean k
              store result in hashtable
              return result
        call fetch-cell-val-helper on the keys task
        clean initial key
        store result in hashtable
        return final result
  action()
```

Following the implementation of the optimizations, the previous example can be revisited with the same calc chain and no previous spreadsheet builds. Now, rather than calling fetch, eval is used while executing the calc chain. The rebuilder first checks A1, since A1 and the parameters of the SUM function are all dirty, 1001 fetch calls will be made as before. After fetching A1 the cell is cleaned. Next up is A2. A2 is dirty as it is newly entered, so the evaluation checks if A2's dependencies are dirty. Since A2's only dependency is two A1s the builder realizes that there is no need to fetch A1 again as now A1 is clean. Therefore to evaluate A2, three fetch calls are made. In total, with the optimized version, the calc chain can be evaluated with 1004 fetches compared to the 3004 fetches before.

4 Experimental Setup and Results

The goal of this section is to describe the experiments conducted by the researcher and display the results of the experiments on the four aspects of using EHOP: code readability, experience of developing and maintaining an application, runtime and memory characteristics. This section is divided into these four aspects where each subsection contains the experimental setup done, the results of its respective aspect and comparisons against functional programming (FP) and/or object oriented programming (OOP).

4.1 Readability

In this section, the results of the readability analysis on MiniExcel and the Excel application by Mokhov et al. [13] will be displayed. Then the results of both applications will be compared to each other. Finally, subjective aspects on how

EHOP produces readable code will be discussed. Table 1 shows the readability metrics measured in both applications where BSC is an acronym for "Build System a la Carte", the article of Mokhov et al. [5]. Detailed explanation of the metrics are given in Section 2.2.

	LOC	HV	E	SMCR
MiniExcel	267	12286	6.18	1.37e-130
BSC	315	13582	6.57	5.72e-145

Table 1: The measurements of objective readability metrics on both MiniExcel and BSC where LOC, HV, E, SMCR is lines of code, Halstead's Volume, Entropy and value calculated from Posnett et al.'s model [10] respectively

As evident from Table 1, one might notice that the SMCR value of both applications seem unusually low, considering SMCR values can be between 0 and 1. This is because SMCR is tailored to work with code snippets rather than full applications. Therefore, other metrics used in Table 1 are more relevant compared to the SMCR value that uses these metrics. Posnett et al. [10] also agrees and states that for applications exceeding 250 lines the SMCR value would be inconclusive. Nonetheless, from SMCR's formula, we can derive that: entropy and Halstead's Volume lower readability while lines of code increases readability. In conclusion, due to the overall difference between HV and E being larger than the difference of LOC, MiniExcel's code was measured as more readable.

Having a quantitative metric of readability is useful. However, due to the subjective and contextual nature of readability the measurements are not fully reliable. Therefore, a subjective comparison must be made on aspects of EHOP in MiniExcel and functional programming in BSC as well as other paradigms that make code readable.

Documentation, in-code or in a separate file, plays a major role in code readability especially for developers that did not participate in the implementation of the application. An aspect that allows to produce clear documentation in EHOP is the ability to define operations separate from their implementations. In MiniExcel, this ability is used to add documentation in effect definitions that explain what an operation does. This allows the developer that is reading it to focus on the operation's documentation rather than passing through implementation details. Moreover, if a developer wants to learn more about an operation (and how it is implemented), they can visit the handler(s) that implements it. For example, in MiniExcel this is done as follows:

```
pub effect dirty-bit
  /*
    Given a key string as input returns True
↪   if the key is dirty i.e the cell it points
↪   to or its dependencies have changed
  */
  fun is-dirty(k : string) : bool
  /*
    Given a key as input cleans the key's cell
  */
  fun clean(k : string) : ()
```

From the code snippet of MiniExcel, it can be seen that compact and easily readable documentation on operations can be written using EHOP because it is not necessary to implement a function right below it. Moreover, operations are defined under an umbrella term such as "dirty-bit" to provide even more readability, as a developer is also informed on the concept of which the operations are implemented for.

Compared to BSC that uses functional programming (FP), a similar ability can be observed. In BSC and in FP with Haskell, one writes a function definition that indicates the name, parameters and return type of the function. Even though function definitions can be far away from their implementation, unlike EHOP, in FP it is common practice to write the implementation of the function right below the definition. An example of such case from BSC is shown below:

```
-- | Read the hash of a key's value. In some cases
↪  may be implemented more
-- efficiently than @hash . getValue k@.
getHash :: Hashable v => k -> Store i k v -> Hash v
getHash k = hash . getValue k
```

In conclusion, EHOP's ability to separate operation definition from implementation and its ability to define operations under umbrella terms gives the edge to EHOP in terms of in-code documentation writing. One might notice the drastic difference between the code snippets which leads to the next point.

Readability is contextual, it depends on factors such as the reader's knowledge or even, depending on the day, their exhaustion. A code snippet can be generally more readable than another code snippet if the snippet is closer to common knowledge. The languages that support object oriented programming (ie. Java [14], C++ [15]) are the most widely used and known amongst other languages [16]. Therefore, a safe assumption can be made that most of the developers in the world know object oriented programming more than functional programming. An example of a function written in object oriented programming is as follows:

```
public static void someFunc(int someArg) {
 // Implementation ...
}
```

The snippet above displays similarities with MiniExcel and dissimilarities with BSC in terms of function definition which proves that the code of MiniExcel is significantly closer to object oriented programming. Since MiniExcel is closer to object oriented programming, it can be concluded that if two code snippets of EHOP and FP were to be shown to a randomly selected group of developers, more of them would pick the code written in EHOP to be more readable. However, this does not change the fact that implementing functions and defining them are significantly more compact and concise (less lines used) in functional programming.

To conclude the readability analysis and comparison, according to Posnett et al.'s model [10], MiniExcel is more readable than BSC. However, conclusions cannot only be drawn from objective models. Subjective aspects of EHOP such as ability to write in-code documentation and relevance of EHOP with knowledge known by most developers gives the edge to EHOP over functional programming in readability.

4.2 Development and Maintenance Experience

This section is separated into three subsections where positive and negative experiences of developing with EHOP are described, followed by a comparison of these experiences with previous experiences with traditional paradigms.

Positive Experiences

Throughout the development and maintenance of the application, EHOP's ability to separate concerns was the main source of positive experiences.

Separation of concerns allowed seamless implementation of updates and improvements to MiniExcel. For example, to add the dirty bit rebuilder into the application, defining a dirty-bit effect and its operations, implementing the operations under a new handler and combining the new handler with the rest was enough. No change was required in other handlers, simply a new handler was implemented and combined with the rest of the handlers. Therefore, separation of concerns achieved by EHOP helped significantly throughout the development and maintenance of the application.

Second positive experience was the ability to define effects and operations under the effect without the need of fully implementing them. This ability allowed for quick development of multiple MiniExcel prototypes. In addition, when a prototype was picked to implement, no significant time was lost as each prototype was written in a short amount of time. This advantage was especially helpful in the early development stages of the application where decisions had to be made on the operations regarding cell storing, cell evaluation and spreadsheet visualisation logic.

In conclusion, EHOP's ability to separate concerns into compact, independent sections of code vastly improved the development and maintenance experience of the researcher.

Negative Experiences

Throughout the development and maintenance of MiniExcel, the catalyst of negative experiences was EHOP's recency. Since EHOP is new, languages that have built-in support for EHOP are underdeveloped. Even an actively maintained and improved language like Koka [6; 7] lacks support in terms of libraries, error messages and external tools. Even though the negative experiences are from developing with Koka, Koka is the language that can best represent using EHOP, as Koka is currently the most widely used amongst the few languages that support EHOP natively.

One aspect of EHOP's underdevelopment was the lack of libraries. Libraries, built-in or third party, are key for any language. Python [17] is possibly the leading language when it comes to the vast library pool, which is one of the main reasons why it is one of the most loved and commonly used languages in 2021 [16]. The lack of libraries in Koka caused few inconveniences throughout the development and maintenance of MiniExcel. An example was that Koka did not have a library for hashtables. As a result, the researcher had to implement a hashtable library from scratch, which took considerable time from the actual application implementation. Therefore in the current state of EHOP, the lack of libraries caused a negative development experience.

Another frustrating aspect of developing with EHOP was effect handling. Since the logic of EHOP is different from traditional paradigms, it entails a learning curve. This learning curve caused some frustrating and confusing errors because the researcher had trouble finding a proper method to handle effects. Furthermore, due to the unique nature of EHOP, foundational concepts that work in traditional paradigms did not work in Koka. For example, the initial plan for the handler of the fetch effect was to call fetch internally with the expectation of the recursive call to be handled within the same handler. However, the expectation fell short as the fetch effect was raised outside the handler which caused the program to fail. To solve this problem, helper functions were implemented and the same handler was used multiple times. The requirement of workarounds and unique functionality caused some frustrating and confusing errors during development and overall lengthened the development process.

In conclusion, working with EHOP's effects can be frustrating and the lack of development in the EHOP ecosystem causes inconveniences. However in terms of underdevelopment, with the promising idea behind EHOP, it is just a matter of time until EHOP's ecosystem becomes a fully functional, production ready and extensive ecosystem.

Comparison

Following the analysis of the development and maintenance experiences using EHOP, this section compares the results with my previous experiences of using the object oriented programming paradigm.

Most positive experiences were sourced by EHOP's ability to provide separation of concerns. Object oriented programming (OOP) also helps to achieve separation of concerns, which is one of the reasons why it is considered as a traditional paradigm. However, previous experience of using OOP with Java [14] suggests that achieving separation of concerns using EHOP is more intuitive than with OOP. For example, when using EHOP the researcher quickly had an idea on how concepts would be separated into different effects. In contrast, when using OOP in other projects it significantly took more time to design an application that would achieve some level of separation of concerns. Therefore, according to the researcher, achieving separation of concerns was simpler and more intuitive when using EHOP compared to using OOP.

Another positive experience was how quickly the researcher was able to develop templates because EHOP allowed them to define operations without implementing them. This concept is nearly identical to object oriented programming's "interface" concept. In OOP, one can define an interface which contains functions that are not yet implemented. Interfaces serve as skeletons to classes that implement them and their functions. In previous OOP projects similar to EHOP, by using interfaces the researcher developed prototypes quickly. OOP's interface is EHOP's effect definition and OOP's interface implementation is EHOP's effect handling. Due to this equality, the experiences are the same when using EHOP and OOP in terms of developing templates.

The EHOP ecosystem is underdeveloped which caused negative experiences throughout development. In contrast, these problems are non-existent in programming languages

that support traditional programming paradigms, as they simply have existed longer, have larger communities and more users. Due to this fact, the researcher never had similar negative experiences when using object oriented programming

The final negative experience was the fact that some concepts of EHOP were different compared to the researcher's knowledge of traditional paradigms which introduced a learning curve. The learning curve was steeper when using EHOP. A reason could be that the researcher learned traditional paradigms in a structured university environment and EHOP by themself. Therefore, in terms of a learning curve, it is not fair to compare experiences even though traditional paradigms were easier to learn.

4.3 Runtime and Memory Characteristics

In this section, the experimental setup utilized to measure runtime and memory will be explained. Then, the results of the runtime and memory analysis on MiniExcel and the Excel application by Mokhov et al. [13] will be displayed.

As mentioned in Section 2.2, different sets of commands were executed in both applications. The commands consist of decimal, single and nested function entries. The measurements are done in a MacOS environment, for MiniExcel Koka's [6] built-in runtime and memory measurement flag "–showtime" is used. Similarly for Mokhov et al.'s application "BSC", Haskell's [8] built-in measurement tool RTS is used. Table 2 displays the measurement results.

	AVG-Elapsed	AVG-Usr	AVG-Sys	AVG-RSS
MiniExcel	0.0004s	0.0011s	0.0024s	2896kb
BSC	0.0002s	0.001s	0.0013s	7836kb

Table 2: Runtime and memory measurements of MiniExcel and BSC.

Before analyzing Table 2 the names of the columns should be explained. All columns are averages of ten measurements done for each application. "Elapsed" refers to the difference between the start time and the end time of the application. "Usr" refers to the time it took to execute the library code. "Sys" refers to the time it took for the execution of kernel operations. Finally "RSS" or "Resident Set Size" refers to the total memory allocated in the RAM.

As evident from Table 2 BSC, is significantly faster than MiniExcel but MiniExcel utilizes less memory. In conclusion, in terms of memory and runtime, for applications that are handling user input without the need of high execution speed, EHOP and functional programming would both be suitable. However, for applications similar to BSC that aim to pre-initialize and evaluate spreadsheets in code for educational purposes, functional programming should be used.

5 Responsible Research

This research assesses effect handler oriented programming (EHOP) on factors that are mostly subjective which affects the reproducibility and the ethical aspects of the research. As for reproducibility, the methodology practiced to retrieve every result is explained, yet a researcher that wants to reproduce this research will not attain the exact results because some results depend on the researcher. For example, development and maintenance experience solely depends on the researcher and it is impossible for two different researchers to have the same experience when developing an application. Ethically, the subjectivity of certain aspects may introduce subconscious experimenter bias where the researcher may have unknowingly disregarded or underestimated the importance of certain measurements that goes against EHOP.

6 Discussion

In this research readability and developer experience were measured and presented in an objective manner. The analysis done on these subjective aspects should not be taken as a fact, rather as a glimpse of effect handler oriented programming's potential. Runtime and memory characteristics should also be considered in a similar manner due to some differences between the compared applications. The program written using functional programming (BSC) was faster and MiniExcel utilized less memory. However, MiniExcel has the extra computation of processing user input while BSC does not which could affect runtime. Moreover Koka [6], the language of MiniExcel, is "not production ready" [6] unlike the language used in BSC, Haskell [8]. In addition, Koka uses Perceus Reference Counting [18] which compared to Haskell's garbage collection, allows Koka to use less than half the memory Haskell would have used [7]. Such differences in both languages could have possibly affected the results.

7 Conclusions and Future Work

The goal of this research was to assess a new programming paradigm, effect handler oriented programming (EHOP) by answering the main research question: "How does using EHOP for implementing a data processing application affect the modularity, readability and maintainability of code compared to traditional paradigms?". To answer this question, EHOP was compared with functional programming and object oriented programming on readability, development and maintenance experience, runtime characteristics, and memory characteristics.

EHOP has shown to produce more readable code than functional programming. It is important to note that readability is both subjective and contextual. It depends on the reader's previous experience, knowledge and their idea of a readable program. Therefore, the conclusion that EHOP produces more readable code than functional programming is not universal but is the case for most developers.

Moreover, development and maintenance experience of EHOP, compared to object oriented programming had its positives and negatives. Briefly, developing and updating an application was easier with EHOP, but the lack of development in the EHOP ecosystem caused frustration and extra work not related to the application. In the future as the EHOP ecosystem grows, the lack of development will be less and possibly the negative sides of EHOP will diminish.

In terms of runtime and memory characteristics, functional programming was proven to be faster but EHOP utilized less memory. A conclusion was drawn that depending on the application type EHOP or functional programming would be

preferable. For example, for applications that expect user input, processes are bound by the time it takes for the user to enter a command thus EHOP would be preferable. On the other hand, for applications that aim to simply execute code without user interaction, functional programming would be more preferable as it is faster.

In conclusion, the answer to the main research question is that EHOP was able to produce more readable, modular and maintainable code compared to traditional paradigms. However, the lack of development in the EHOP ecosystem is currently a bump on the road for EHOP. In the future as the EHOP ecosystem grows and the lack of development decreases, EHOP will likely have a spot amongst the traditional paradigms.

This research implements an open source Excel-like application "MiniExcel" from scratch that contributes to the EHOP ecosystem. It also introduces interesting ways to utilize EHOP to implement certain concepts. Moreover, using EHOP it defines "Hierarchical EHOP", a new pattern to develop a hierarchical structure of concepts which enforces certain rules on the interaction of different concepts and produces a easier to read program structure.

On a final note, this research leaves space for improvements and future work. An improvement that would enrich the comparisons is to implement the exact application in other paradigms rather than using a third party application that implements a similar application. Moreover, since this research only assesses using EHOP for data processing applications, other application types such as games, servers etc. using EHOP should also be assessed. This would not only provide more research on EHOP but it would also add new applications to the EHOP ecosystem.

References

[1] Microsoft, "Microsoft excel spreadsheet software: Microsoft 365," 2022. Last accessed may 2022. Available: https://www.microsoft.com/en-us/microsoft-365/excel.

[2] Microsoft, "Excel functions," 2022. Last accessed june 2022. Available: https://support.microsoft.com/en-us/office/formulas-and-functions-294d9486-b332-48ed-b489-abe7d0f9eda9.

[3] D. Hillerström, *Foundations for Programming and Implementing Effect Handlers*. PhD thesis, The University of Edinburgh, 2021.

[4] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[5] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," *Proc. ACM Program. Lang.*, vol. 2, jul 2018.

[6] D. Leijen, "The koka programming language," 2022. Last accessed may 2022. Available: https://koka-lang.github.io/koka/doc/book.html.

[7] D. Leijen, "Koka: a functional language with effects," 2022. GitHub repository. Available: https://github.com/koka-lang/koka.

[8] S. Marlow *et al.*, "Haskell 2010 language report," 2010. Available: https://www.haskell.org/onlinereport/haskell2010/.

[9] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do," *SIGPLAN Not.*, vol. 52, p. 500–514, jan 2017.

[10] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," *Proceedings - International Conference on Software Engineering*, pp. 73–82, 2011.

[11] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[12] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018. e1958 smr.1958.

[13] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," 2020. GitHub repository. Available: https://github.com/snowleopard/build.

[14] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st ed., 2014.

[15] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Professional, 4th ed., 2013.

[16] StackOverflow, "Stack overflow developer survey 2021," 2021. Last accessed may 2022. Available: https://insights.stackoverflow.com/survey/2021.

[17] G. van Rossum, "Python programming language," in *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007* (J. Chase and S. Seshan, eds.), USENIX, 2007.

[18] A. Reinking, N. Xie, L. de Moura, and D. Leijen, "Perceus: Garbage free reference counting with reuse," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 96–111, Association for Computing Machinery, 2021.