Applicability of Effect Handler Oriented Programming (EHOP) for Text-Based Game Development

Ivan Todorov
Supervisor(s): Casper Bach Poulsen, Cas van der Rest, Jaro Reinders EEMCS, Delft University of Technology, The Netherlands
24-6-2022

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 (EHOP) is a recently proposed programming paradigm, which aims to provide a separation of concerns by isolating the handling of side-effects from the main application logic. Nowadays, as the core concepts behind EHOP are being added to more and more programming languages, it is evident that EHOP is slowly but steadily growing in popularity. Therefore, it is important to explore the applicability of EHOP for different areas of software development. However, so far, very little research has been conducted on this topic and, thus, barely any possible application domains of EHOP have been investigated. This study focuses on a potential field of application of EHOP, which has not been covered by previous research, namely - text-based game development, and aims to determine the extent to which the usage of EHOP for text-based game development affects the modularity, readability and maintainability of the source code. This goal will be achieved by performing both a qualitative as well as a quantitative analysis of the source code of a text-based game, written in Koka - a state-of-the-art programming language, which supports EHOP. The results show that there are substantial benefits to using EHOP for text-based game development. It significantly improves the modularity, readability and maintainability of the source code at the cost of very little to no performance.

1 Introduction

Recent developments in the field of Programming Languages have led to the idea of creating high-level programming languages, which provide a separation of concerns. This means that it should be possible to understand and reason about a program's source code independently of orthogonal concerns. As a result, the concept of effect handlers and the possible ways to incorporate it into modern programming languages have become a subject of extensive research. Algebraic effect handlers [1] were originally introduced as a theoretical foundation for studying programming languages with side effects. More recently, however, they have become the core concept behind the programming paradigm, called Effect handler oriented programming (EHOP).[1] The main goal of EHOP is to allow programmers to represent side effect by declaring operations, which are implemented using effect handlers, while said effect handlers are implemented separately from the application code, which uses the operations. This is achieved by introducing the concept of effect operations and defining the following three main components of a program's source code [2]:

- Effect signatures - declarations of effect operations;
- Effectful functions - functions, which use effect operations;
- Effect handlers - functions, which provide implementations of effect operations.

[1]Hereafter: EHOP - Effect handler oriented programming

```
// Emitting messages; how to emit is TBD.
// Just one abstract operation: emit.
effect fun emit(msg : string) : ()

// Emits a standard greeting.
fun hello() : emit ()
  emit("hello world!")

// Emits a standard greeting to the console.
pub fun hello-console1() : console ()
  with handler
    fun emit(msg) println(msg)
  hello()
```

An example[2] of how a program's source code can be split into the aforementioned three parts is presented above. It depicts the declaration, usage and handling of a simple `emit` effect in the Koka programming language [4]. The `emit` effect is declared by the following effect signature:

```
effect fun emit(msg : string) : ()
```

According to this effect signature, the `emit` effect consists of only a single function with the same name, which takes one argument of type `string` and has the unit type (denoted as `()` in Koka) as its return type. Additionally, in the example, shown above, the `hello` function is an effectful function, because it uses the `emit` effect. Finally, the `hello-console1` function, shows how the `emit` effect can be handled using the following effect handler:

```
with handler
    fun emit(msg) println(msg)
```

This effect handler provides a concrete implementation for the `emit` effect by specifying that, when the `emit` function is called with some string as its argument, it will simply print said string to the console. As a result, calling the `hello-console1` function would simply print the string "hello world!" to the console.

As can be seen from the given example, splitting a program's source code into effect signatures, effectful functions and effect handlers, allows developers to isolate the handling of side effects from the main application logic, resulting in a higher level of separation of concerns, compared to "traditional" implementations.[3]

As of the time of conducting this research, there are only a few state-of-the-art programming languages, which support EHOP, such as Koka and Frank [5]. Nevertheless, the core concepts behind EHOP are also available in the widely-used functional programming language Haskell [6], through the usage of libraries for algebraic effects, such as `effect-handlers` [7], `fused-effects` [8] and `polysemy` [9]. This shows that EHOP is gradually gaining popularity as a practical tool for software development. Therefore, it is

[2]This example is taken from Koka's website [3].

[3]Hereafter: traditional implementation - an implementation, which follows either the Object-oriented programming (OOP) paradigm or the Functional programming (FP) paradigm

important to explore the extent to which EHOP is applicable for different areas of software development. However, so far, barely any research has been done on this subject and, thus, very few possible applications of EHOP have been analysed - such as implementing the core of the UNIX operating system [10].

The goal of this study is to reduce the gap in the existing literature by focusing on a potential field of application of EHOP, which has not been covered by previous research, namely - text-based game development, and answering the following question:

When EHOP is used for text-based game development, to what extent does the support for separation of concerns of EHOP affect the modularity, readability and maintainability of the source code, compared to traditional implementations?

The importance of this kind of research stems from the fact that text-based games (and all games, in general) are programs, which have multiple side effects, that need to be managed alongside the main application logic in traditional implementations (e.g. the inputs of the players, the state of the game world, the state of the characters, etc.). This means that, in theory, the separation of concerns, which EHOP provides, should significantly increase the modularity, readability and maintainability of the source code by separating the management of all side effects from the main application logic, making EHOP not only applicable but also preferred for such applications. However, this hypothesis needs to be confirmed in practice.

The contributions of this paper are:

- an explanation of the methodology, used during this research (Section 2);
- a description of the process of developing a text-based game using EHOP and an assessment of the effects of using EHOP for text-based game development on the modularity, readability and maintainability of the source code (Section 3);
- an analysis of the effects of using EHOP for text-based game development on the quantitative aspects of the source code (Section 4);
- a discussion about the ethical aspects of this study and its reproducibility (Section 5);
- a conclusion about the effects of using EHOP for text-based game development on the qualitative and quantitative aspects of the source code and an exploration of possibilities for future work (Section 6).

2 Methodology

In order to answer the main research question of this study, it is essential to evaluate the qualitative aspects of EHOP by performing a qualitative analysis of the readability and experience of developing and/or maintaining text-based games, implemented using EHOP. However, it is possible that any potential benefits of using EHOP for text-based game development come at the cost of performance. Thus, during this study it is crucial to also explore the quantitative aspects of EHOP by performing a quantitative analysis of the run time and memory characteristics of text-based games, developed using state-of-the-art programming languages, that support EHOP (such as Koka [4], Frank [5] or Haskell [6]). Following the reasoning, described above, the main research question can be split into the following sub-questions:

- When EHOP is used for text-based game development, to what extent does the support for separation of concerns of EHOP affect the modularity of the source code, compared to traditional implementations?
- When EHOP is used for text-based game development, to what extent does the support for separation of concerns of EHOP affect the readability of the source code, compared to traditional implementations?
- When EHOP is used for text-based game development, to what extent does the support for separation of concerns of EHOP affect the maintainability of the source code, compared to traditional implementations?
- When EHOP is used for text-based game development, to what extent does the support for separation of concerns of EHOP affect the performance (run time, memory characteristics, etc.) of the source code, compared to traditional implementations?

For the purposes of this research the Koka programming language was chosen, as an example of a modern programming language, which supports EHOP, for the following reasons:

- Koka inherently supports EHOP;
- As of the time of conducting this research, the Koka programming language is getting regular updates;[4]
- Source code, written in Koka, can be compiled and executed on any operating system, making the study independent of a concrete operating system and, therefore, more easily reproducible.

As a result, this study has been implicitly divided into the following five tasks, which should be performed exactly in the specified order:

1. Choose a text-based game, which will be feasible to implement, considering the time constraints, imposed by the duration and deadlines of this research, and implement it in Koka;
2. Perform a qualitative analysis of the readability and experience of developing and/or maintaining the text-based game, in order to evaluate the modularity, readability and maintainability of the source code;
3. Perform a quantitative analysis of the run time and memory characteristics of the text-based game, in order to evaluate the performance of the source code.

Additionally, part of this study was dedicated to exploring different existing metrics for evaluating the qualitative and quantitative aspects of a program's source code and investigating the applicability of said metrics to source code, written in Koka.

[4]For more information, go to https://github.com/koka-lang/koka

When it comes to evaluating the qualitative aspects of a program's source code, the following metrics were explored:

- Buse's model [11] - a readability metric;
- Cyclomatic complexity [12] - a complexity and readbility metric;
- Halstead's model [13] - a complexity metric;
- Module cohesion - a readability and maintainability metric;
- Module coupling - a modularity metric;
- Posnett's model [14] - a readability metric.

However, all of the metrics listed above are either very difficult or even impossible to assess by hand. All of them are usually evaluated using special software quality tools, which are designed and developed with the goal of computing said metrics. Unfortunately, as of the time of conducting this study, no such software quality tools have been found, which have support for the Koka programming language. Therefore, since developing such a tool is beyond the scope of this research and assessing the aforementioned metrics by hand was deemed unfeasible, given the time constraints of this study, it was concluded that said metrics are not applicable to this research. As a result, it was decided that, for the purposes of this study, the most fitting way of exploring the qualitative aspects of EHOP would be through practical examples. Thus, in Section 3, which describes the process of developing a text-based game in Koka, special attention is put on the effects, that are used, and how their addition affects the modularity, readability and maintainability of the source code.

When it comes to analysing the quantitative aspects of EHOP, it was determined that, for the purposes of this research, it would be sufficient to evaluate the following metrics:

- The compilation time of the finished game;
- The start time of the finished game;
- The time, which the game needs, in order to execute a player's turn;
- The peak memory usage during a game.

The reason behind the selection of the metrics listed above is that together they would present a clear picture of both the performance as well as the memory characteristics of a text-based game, written in Koka. However, in order to measure those metrics, it should be possible to simulate a game. This means that instead of the game occasionally waiting for a player's input, all inputs should be specified in the exact order, before the launch of the game, and then the game will just play itself, following the provided inputs. As a result, this was determined to be a required feature of the text-based game, which would be implemented in Koka.

3 Developing a Text-Based Game in Koka

For the purposes of this research a fully-featured shared-screen multiplayer text-based chess game was developed entirely in the Koka programming language [4]. The development process can be implicitly divided into the following stages:

1. Implementing the game's main menu;
2. Implementing the basic components of the game - the two player and the chess pieces;
3. Implementing the chess board;
4. Implementing the players' turns;
5. Implementing the win and draw conditions;
6. Extracting functionality, which should be easy to augment, into side effects;
7. Implementing the feature, which allows a game to be simulated by providing the players' inputs in the exact order, before the launch of the game.

3.1 Implementing the game's main menu

The first task of implementing the text-based chess game was to implement a simple main menu. It was decided, that a main menu, which allowed users to either start a new game or close the application, would be sufficient. If at a later stage more options were needed, the main menu could be extended to provide them. Implementing the basic main menu did not pose any difficult challenges. However, during its development, it was concluded that, since the functionality of the main menu might have to be extended, the handling of the user input should be made easily extendable or replaceable. Therefore, the function, which parses a user's input and, based on it, decides on what action should be taken, was extracted into the following side effect:

```
effect fun read-input-main-menu() : maybe<action>
```

The main goal behind this refactoring was to allow the functionality of the main menu to be expanded by simply creating a new handler for the `read-input-main-menu` effect, or by extending the already-existing handler, but without changing the core logic of the main menu. As a result, this approach would significantly improve the modularity and, by extension, the maintainability of the source code, since it would allow the functionality to be extended or changed by only editing the effect handler.

3.2 Implementing the basic components of the game

When the main menu was completed, the goal was set on implementing the basic components of the game - namely, the two players and the chess pieces. Creating the players was a straight-forward task, which involved defining a data type with two alternative constructors - one for each player.

```
type player
    White
    Black
```

The pieces were, initially, implemented using the same approach - by defining a data type with six alternative constructors - one for each type of piece, all of which required two parameters - the player, who owns the piece, and the location of the piece. It was decided that the locations would be implemented as a tuple of integers, representing the X and Y coordinates of a given location on the chess board, such that the square A1 would be the location (0, 0) and the square H8 would be the location (7, 7). As a result, the initial implementation of the `piece` data type was the following:

```
type piece
    Pawn(owner : player, square : location)
    Knight(owner : player, square : location)
    Bishop(owner : player, square : location)
    Rook(owner : player, square : location)
    Queen(owner : player, square : location)
    King(owner : player, square : location)
```

However, as can be seen from the code snippet above, this initial approach involved code duplication, since all constructors have the exact same set of arguments. This, of course, reduces the maintainability of the source code. An additional drawback was that adding more properties to a certain piece type (as discussed in Section 3.4) would increase the number of arguments, required by its constructor, to three or even more, which reduces the readability of the source code, and should, therefore, be avoided [15]. Thus, it was decided that the piece types should be represented by a separate data type with six alternative constructors - one for each type of piece:

```
type piece-type
    Pawn
    Knight
    Bishop
    Rook
    Queen
    King
```

This way, it would be possible to add specific properties to some types of pieces, which was, indeed, necessary during later stages of the development process (as discussed in Section 3.4), without making the argument lists of their constructors too long. Moreover, this refactoring removed the code duplication, which was part of the initial implementation, because pieces would now be implemented as a `struct` - Koka's name for a data type, that only has a single constructor with the same name as the data type, and would require three arguments - the type of the piece, the player, who owns the piece, and the location of the piece:

```
struct piece
    piece-type : piece-type
    owner : player
    square : location
```

3.3 Implementing the chess board

When the two players and the chess pieces were completed, it was time to implement the chess board. After different ways of representing the chess board were considered, it was decided that a piece-list [16] would be the most suitable solution in the current case. There were two main reasons for this choice:

- In the existing implementation the pieces already contain information about their location on the board. Therefore, a piece centric [17] board representation would be most fitting;
- Koka does not have built-in support for sets, but supports lists.

As a result the chess board was implemented as a `struct`, the constructor of which only requires a single argument - the list of figures, that are initially on the board:

```
struct board
    figures : list<figures>
```

3.4 Implementing the players' turns

The next step of implementing the text-based chess game was to implement the players' turns. This was undoubtedly the most time-consuming part of the entire development process as well as the most logically complex one. However, the difficulty came mostly from the large number of possible moves in chess and the interactions between them. While the basic moves were fairly straight-forward to implement, things like castling, pawn-promotion, en-passant and the fact that, if a pawn has not moved yet, it can move two squares forward, rather than just one, required some adjustments to be made to the `piece-type` data type. As a result, the `piece-type` data type was changed to the following:

```
type piece-type
    Pawn(has-moved : bool, en-passant : bool)
    Knight
    Bishop
    Rook(has-moved : bool)
    Queen
    King(has-moved : bool)
```

The `has-moved` argument, which was added to pawns, rooks and kings, denotes whether the figure has been moved, while the `en-passant` argument, which was added only to pawns, indicates if a pawn can be captured by an en-passant move.

The only other noteworthy aspect about this task is that, similarly to the main menu, it was decided that all functions, which parse and process user input, should be extracted into separate side effects. The reason behind this decision, once again, was that, if, at a later stage, additional input options need to be added or an existing input option has to be changed, this could be done by simply editing the effect handler for the given side effect. This refactoring, which significantly improved the modularity and, by extension, the maintainability of the source code, resulted in the creation of the following three side effects:

- `read-input-turn`: This side effect reads and processes a player's input, when it is their turn. The possible actions are the following:
  - Move a piece: The format, in which piece moves are entered, is very straight-forward - the start square followed by the end square. For example, the input "B2B4" indicates that a figure should be moved from square B2 to square B4. The game then checks, if the current player has a figure at square B2 and whether moving said figure from B2 to B4 is a valid move. If so, then the move is performed. Otherwise, the input is deemed invalid and the current player is asked for a new input.
  - Perform a castling: A kingside castling is denoted by "0-0" or "O-O", while a queenside castling is denoted by "0-0-0" or "O-O-O".

  - Offer a draw: A player can offer a draw by using one of the following inputs: ”Draw”, ”Tie”, ”Remis”.
  - Surrender: A player can surrender by using one of the following inputs: ”Surrender”, ”Good game”, ”GG”, ”Forfait”, ”FF”.

- `read-input-pawn-promotion`: This side effect reads and parses a player’s input, when a pawn promotion occurs. In such situations the player, whose pawn is being promoted, can decide, if they want the pawn to be replaced by a knight, a bishop, a rook or a queen.
- `read-input-draw-offer`: This side effect reads and parses a player’s input, when their opponent offers a draw. The current player can either accept or reject the draw offer. If a draw offer is rejected, it is still the opponent’s turn. However, they cannot offer a draw again on the same turn - only one draw offer is possible per turn.

3.5 Implementing the win and draw conditions

When the players’ turns were implemented, it was time for the final part of the actual game to be added - the win and draw conditions. The following two win conditions were implemented:

- Checkmate: If the player, whose turn it currently is, is in check and has no valid moves, then they are in checkmate and their opponent wins the game.
- Surrender: If a player surrenders, their opponent wins the game.

While the win conditions, listed above, were fairly simple to implement, the draw conditions were substantially more complex. Four draw conditions were implemented:

- Stalemate: If the player, whose turn it currently is, has no value moves and they are not is check, the game is declared a draw.
- Accepting a draw offer: If a player accepts a draw offer, made by their opponent, the game is game declared a draw.
- A variation of the threefold repetition rule [18]: If the same position occurs three times during a game, the game is automatically declared a draw. In this context two positions are said to be the same, if ”the same types of pieces occupy the same squares, the same player has the move, the remaining castling rights are the same and the possibility to capture en passant is the same” [18].
- A variation of the fifty-move rule [19]: If during the last fifty moves no pieces have been captured and no pawns have advanced, the game is automatically declared a draw. In this context one move consists of both players performing a turn.

While the former two of the aforementioned draw conditions were straight-forward to implement, the latter two required the tracking of the game’s history of positions. At the beginning of a game the history is, as expected, empty. Then, at the end of each player’s turn the current position is added to the history. However, whenever a pawn advances or a piece is captured, the history is cleared.

This was implemented using Koka’s built-in `st<h>` effect, which is an alias for the combination of the `heap<h>` effect, the `read<h>` effect and the `write<h>` effect. The usage of the `st<h>` effect makes it possible to store variables in the heap and use them by reference. Therefore, when said variables are passed as arguments to other functions, they are passed by reference, instead of by value, which is how standard variables are passed in Koka. As a result, multiple functions can manipulate the exact same variable by changing the value, stored at the corresponding location in the heap.

This usage of side effects clearly demonstrates how the features of EHOP can be used to easily extend the existing functionality of the application in a very modular way - something, which would not have been as simple to achieve in traditional implementations.

3.6 Extracting additional functionality into side effects

After the entire chess game was fully-functional it was decided that functionality, which is likely to be changed in the future, should be separated from the main application logic and turned into side effects. The goal behind this decision was to increase the modularity and the maintainability of the source code, because if, at a later stage, someone wants to change said functionality, they could do so by simply editing the corresponding effect handler, without delving into the main application code.

The only functionality, which was determined to be likely to change, was the printing of the chess board. The current implementation prints the chess board in a very plain and basic way. Therefore, it was concluded that, in the future, a fancier way of printing the chess board could be implemented and this should be possible without changing the remainder of the source code. As a result, the function, responsible for printing the chess board, was extracted into a separate side effect and its implementation was turned into a handler for said side effect:

```
effect fun show-board(x : board) : string
```

3.7 Allowing games to be simulated

The last part of the development process of the text-based chess game was to implement the functionality, which would allow games to be simulated. This means that, instead of the game waiting for the players’ inputs, those inputs should be provided in the exact order, before the game is launched. As discussed in Section 2, this feature was required, in order to conduct the performance analysis of the source code. Therefore, it was decided that it should also be possible to run a simulated game, which does not print things to the console. This would make it possible to assess the performance of the code, which handles the application logic, without the additional time, required for printing.

In order for this goal to be achieved, a new side effect was created, which consisted of three functions - one which handles the players’ inputs, one which prints some text to the console and one which prints a line of text to the console.

```
effect execution-method
    fun user-input() : string
    fun custom-print(x : string) : ()
    fun custom-println(x : string) : ()
```

Additionally, the following three handlers for this side effect were created:

- `normal-game`: This handler is the one, which would be used in a normal game between two players. The implementation allows the players to provide their inputs during the game and everything is printed to the console as usual.
- `simulation-with-printing`: This handler would be used to simulate a game, but without disabling the printing of text to the console. The implementation requires the players' inputs to be provided in the correct order, before the game is launched. However, everything is still printed to the console as usual.
- `simulation-without-printing`: This handler would be used to simulate a game with the printing of text to the console disabled. The implementation requires the players' inputs to be provided in the correct order, before the game is launched and no text is printed to the console during the simulation of the game.

4 Experimental Setup and Results

After the text-based chess game was developed and the effects of using EHOP on the qualitative aspects of its source code were assessed, the quantitative aspects of said source code had to be analysed. For this purpose, as discussed in Section 2, the following metrics were used:

- The compilation time of the finished game;
- The start time of the finished game;
- The time, which the game needs, in order to execute a player's turn;
- The peak memory usage during a game.

All of the metrics, listed above, were evaluated on a MSI GS73 Stealth 8RF laptop, which had an Intel® Core™ i7-8750H CPU, 32GB RAM and a Nvidia GeForce GTX 1070 Max-Q GPU and was running the Ubuntu 22.04 operating system. The evaluation was done using the `time` command, which is available by default on Ubuntu 22.04 as part of the `time` package and can be executed using `/usr/bin/time`. It is important to note that the `time` command measures three different notions of time, all of which were analysed during this study:

1. Real time - "Elapsed real (wall clock) time used by the process, in [hours:]minutes:seconds" [20];
2. User time - "Total number of CPU-seconds that the process used directly (in user mode), in seconds" [20];
3. System time - "Total number of CPU-seconds used by the system on behalf of the process (in kernel mode), in seconds" [20].

Additionally, it is noteworthy that, when using the `time` command, the peak memory usage of the program is displayed as the "maximum resident set size of the process during its lifetime, in Kilobytes" [20].

Finally, in order to properly evaluate the effects of using EHOP for text-based game development on the quantitative aspects of the source code, it was necessary to compare the aforementioned quantitative metrics between an implementation which uses side effects and one which does not. For this purpose, the existing source code of the text-based chess game was refactored, with the goal of removing all custom side effects from it. As a result, the new implementation only uses the effects, which are built into Koka [4]. Then, the quantitative metrics, discussed above, were measured for both the original implementation as well as the new one and the results were compared to each other.

It has to be noted that this is not an ideal comparison, since the new implementation still uses some side effects, namely - those which are built into Koka. However, the only way to compare the initial implementation to one which does not use any side effects would be to implement the text-based chess game in a different programming language. This approach was deemed unfeasible, given the time constraints, imposed by this research, and, therefore, the aforementioned procedure was chosen.

Using the methodology, described above, the compilation time of the finished game was measured as the average of ten compilations. Each compilation was performed using the command:

```
koka -o2 -o main main.kk
```

The results of this experiment are depicted in Figure 1.

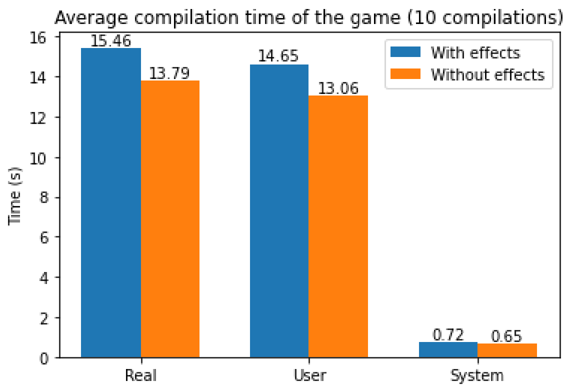

Figure 1: The average compilation time of the finished text-based chess game with and without the usage of custom effects.

Figure 1 shows that the implementation of the text-based chess game, which uses custom side effects, took slightly longer to compile than the implementation, that does not use custom side effects. Moreover, it can bee seen that, while the System time, used to compile the two implementations, was very close, the addition of custom side effects brought a more noticeable increase of about 12% to both the User time and, most importantly, the Real time. The most likely reason for this increase in compilation time is that the custom side effects were implemented as different modules in separate files from the remainder of the source code. Therefore, during the compilation process of the entire game, there were more files and modules, which had to be compiled, in the implementation, that uses custom effects. As a result, its compilation took longer than that of the implementation, which does not use any custom effects.

When it comes to the start time of the finished game, that was measured using an "empty" simulation - a simulation in which, as soon as the main menu is loaded, the application is immediately closed by providing the corresponding input. This way, the time spent, while the application is running, will be negligible and, therefore, the start time of the application will have the greatest impact on the final measurement. The start time of the finished game was measured as the average of one hundred "empty" simulations and the results are shown in Figure 2.

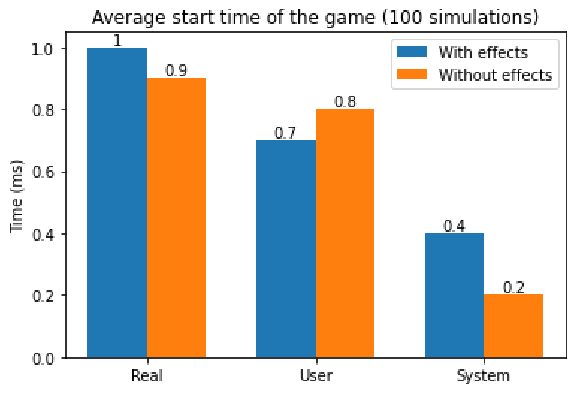

Figure 2: The average start time of the finished text-based chess game with and without the usage of custom effects.

From Figure 2 it can be seen that there is no substantial difference in starting time between the two implementations. While the implementation which uses custom side effects requires slightly less User time, it takes marginally more System time and Real time. However, the differences between the two implementations are so small that the most likely reason for them is just run-to-run variance.

In order to measure the time needed to execute a player's turn, two professional games of chess were simulated, namely - the shortest decisive, non-forfeited game and the longest decisive game, played during a World Chess Championship. It is important to note that a game is decisive, when it does not end in a draw and, therefore, has a determined winner.

"The shortest decisive, non-forfeited world championship game occurred between Viswanathan Anand and Boris Gelfand in game eight of the World Chess Championship 2012" [21]. In this game both players played seventeen turns. Therefore, in order to approximate the time needed to execute a player's turn in a simulation of this game, the total run time of the simulation has to be divided by thirty-four, since there were thirty-four individual turns during this game.

The longest game played in a world championship, which also happens to be decisive, took place in game six of the World Chess Championship 2021 between Magnus Carlsen and Ian Nepomniachtchi [21]. This game consisted of both players playing one hundred and thirty-six turns each. Therefore, in order to approximate the time needed to execute a player's turn in a simulation of this game, the total run time of the simulation has to be divided by two hundred and seventy-two, since there were two hundred and seventy-two individual turns during this game.

The average time for executing a player's turn in both games, described above, was computed as the average of one hundred simulations of those games. Moreover, said average time was measured for both the implementation which uses custom effects as well as the one, that does not. The outcomes of this experiment for the shortest decisive, non-forfeited game and the longest decisive game, played in a World Chess Championship, are shown in Figure 3 and Figure 4, respectively.

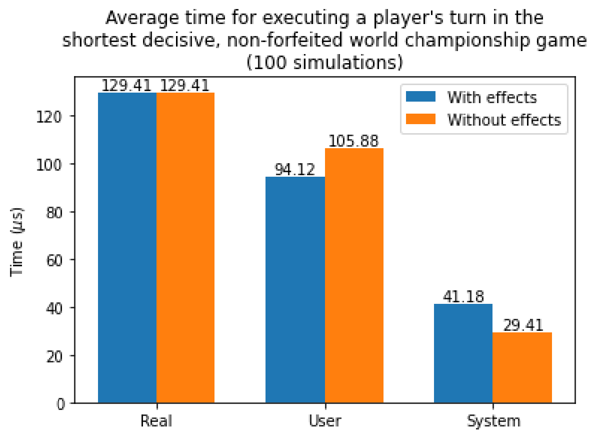

Figure 3: The average time needed to execute a player's turn in a simulation of the shortest decisive, non-forfeited game, played during a World Chess Championship, with and without the usage of custom effects.

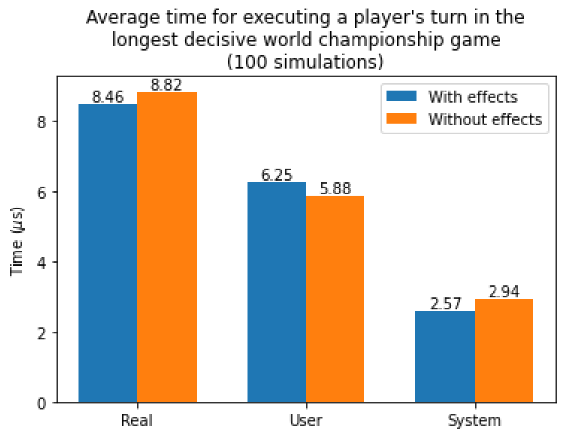

Figure 4: The average time needed to execute a player's turn in a simulation of the longest decisive game, played during a World Chess Championship, with and without the usage of custom effects.

From Figure 3 and Figure 4 it can be seen that there is no significant difference in any of the measured times between the implementation of the text-based chess game which uses custom effects and the one, that does not. In fact, the differences in the timings between the two implementations are so marginal that they are most certainly caused by run-to-run variance. This shows that the addition of custom effects does not influence in any way - either positive or negative, the average time needed to execute a player's turn.

The peak memory usage was evaluated for both of the aforementioned professional chess games by measuring the maximum resident set size during one hundred simulations of said games. The outcomes of this experiment are depicted in Figure 5.

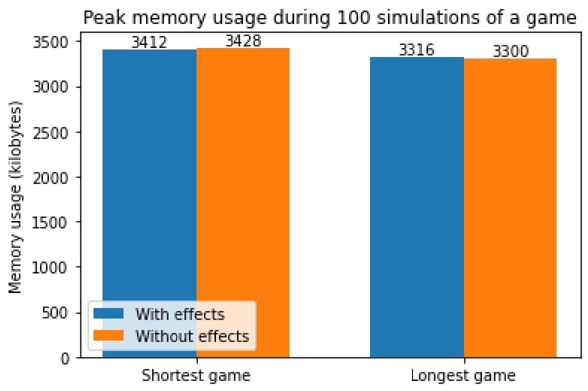

Figure 5: The peak memory usage measured during one hundred simulations of the shortest decisive, non-forfeited game and the longest decisive game, played during a World Chess Championship, with and without the usage of custom effects.

The results in Figure 5 clearly show that there is no significant difference in peak memory usage between the implementation which uses custom effects and the one, that does not. This means that the addition of custom effects does not bring any drawbacks in terms of peak memory usage.

Overall, the quantitative analysis, performed during this study, reveals that the usage of EHOP for text-based game development has almost no downsides with regard to the quantitative aspects of the source code. The only metric, which was noticeably and consistently negatively influenced by the addition of custom effects, was the compilation time of the game's source code. The results of all other quantitative metrics, which were evaluated, showed that said metrics were not influenced by the addition of custom effects in any significant way whatsoever.

5 Responsible Research

The only ethical issue, which arises from this study, is that of subjectivity, namely - the subjectivity of the assessment of the qualitative aspects of the source code. As discussed in Section 2, as of the time of conducting this research, there are no software quality tools which support the Koka programming language [4]. As a result, there was no way to objectively evaluate the modularity, readability and maintainability of the source code of the text-based chess game. Therefore, the assessment of the effects of using EHOP for text-based game development on the qualitative aspects of the source code was performed in a mostly, if not entirely, subjective way. This means that any conclusions, which are drawn from this assessment, should be taken with a grain of salt, as it is entirely possible that the subjective evaluation was influenced either positively or negatively by the researcher's personal experience with EHOP. Additionally, this subjectivity makes the qualitative analysis, performed during this study, absolutely not reproducible, since it depends on the researcher, who performed the evaluation, and their own biases.

On the other hand, the quantitative analysis, which was conducted as part of this research, is entirely reproducible, if the correct hardware is available. The reason for this is that both the source code of the text-based chess game as well as all scripts, which were used during the quantitative analysis, are open-source and publicly available on GitHub.[5] Moreover, the entire methodology, used to evaluate the quantitative metrics of the source code, was explained in Section 4 of this paper. Finally, the Ubuntu 22.04 operating system is also free and open-source and can be downloaded from the website of Canonical - the company behind Ubuntu.[6] As a result, if one has the exact same hardware, which was used during this study, they can, first, download and install the Ubuntu 22.04 operating system from Canonical's website, then download and install Koka by following the instructions on Koka's website [3] and, finally, using the source code of the text-based chess game as well as the scripts from GitHub, they can follow the methodology, described in Section 4, in order to perfectly reproduce the quantitative analysis, performed during this research.

6 Conclusions and Future Work

The aim of this study was to explore the extent to which the usage of EHOP for text-based game development affects the modularity, readability and maintainability of the source code. In order to achieve this goal, a qualitative analysis of the source code of a text-based chess game, written entirely in Koka [4] - a state-of-the-art programming language, which supports EHOP, was performed. Additionally, a quantitative analysis of said source code was conducted, in order to inspect, whether the usage of EHOP has any substantial drawbacks in terms of performance.

The results of the qualitative assessment showed that, while the features of EHOP only mildly enhance the readability of the source code, they significantly improve its modularity and, by extension, maintainability. Moreover, the outcome of the quantitative analysis revealed that EHOP brings all those benefits without any negative consequences to the start time, run time and peak memory usage of the application. The only downside of EHOP is that it slightly increases the compilation time of the program.

In conclusion, this research demonstrates that there are substantial benefits to using EHOP for text-based game devel-

[5]For more information, go to: https://github.com/ivanstodorov/chess-game-koka

[6]For more information, go to https://ubuntu.com/

opment. It almost allows game developers to have their cake and eat it too. As the results illustrate, EHOP brings remarkable improvements to the qualitative aspects of the source code, namely - modularity, readability and maintainability, at the cost of very little to no performance.

However, this research has many limitations, which can be addressed by future studies. Firstly, the lack of available software quality tools, which support the Koka programming language, forced the qualitative analysis of the source code to be largely, if not entirely, subjective. Therefore, a future study can focus on developing a software quality tool, which supports Koka. Then, it would be possible to perform an objective qualitative analysis of the source code of the text-based chess game, developed as part of this research.

Another possibility for future work would be to develop the exact same text-based chess game in either an object-oriented or functional programming language and compare that new implementation to the one in Koka in both qualitative and quantitative aspects. This would result in a much better comparison than the one performed during this study.

It is also possible that a future study focuses on the exact same topic, namely - the applicability of EHOP for text-based game development, but assesses another programming language, which supports EHOP, such as Frank [5] or Haskell [6]. The reason, why such research would be very important, is that it could compare the advantages and disadvantages of the different existing implementations of effect handlers, such as Koka's effect handlers, Frank's effect handlers, scoped effect handlers [22], etc. As a result, such an approach could lead to significantly different results than the ones obtained during this research.

Finally, future studies can and should focus on exploring the applicability of EHOP for other areas of software development, other than text-based game development. The reason for this is that text-based game development is a very small part of the entire software development field. Therefore, even if EHOP is amazing for text-based game development, which, according to the results of this research, it absolutely is, that does not necessarily make it applicable for any other software development tasks.

References

[1] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," in *European Symposium on Programming*, pp. 80–94, Springer, 2009.

[2] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effect handlers for the masses," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[3] "The Book of Koka: The Koka Programming Language." https://koka-lang.github.io/koka/doc/book.html. Accessed: 2022-05-30.

[4] D. Leijen, "Type directed compilation of row-typed algebraic effects," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499, 2017.

[5] L. Convent, S. Lindley, C. McBride, and C. McLaughlin, "Doo bee doo bee doo," *Journal of Functional Programming*, vol. 30, 2020.

[6] "Haskell: An advanced, purely functional programming language." https://www.haskell.org/. Accessed: 2022-06-14.

[7] "effect-handlers: A library for writing extensible algebraic effects and handlers." https://hackage.haskell.org/package/effect-handlers. Accessed: 2022-06-14.

[8] "fused-effects: A fast, flexible, fused effect system." https://hackage.haskell.org/package/fused-effects. Accessed: 2022-06-14.

[9] "polysemy: Higher-order, low-boilerplate free monads." https://hackage.haskell.org/package/polysemy. Accessed: 2022-06-14.

[10] D. Hillerström, "Composing UNIX with Effect Handlers: A Case Study in Effect Handler Oriented Programming," in *ML Workshop*, 2021.

[11] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 121–130, 2008.

[12] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[13] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[14] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*, pp. 73–82, 2011.

[15] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[16] "Piece-Lists - Chess Programming Wiki." https://www.chessprogramming.org/Piece-Lists. Accessed: 2022-06-07.

[17] "Board Representation - Chess Programming Wiki." https://www.chessprogramming.org/Board_Representation. Accessed: 2022-06-07.

[18] "Threefold repetition - Wikipedia." https://en.wikipedia.org/wiki/Threefold_repetition. Accessed: 2022-06-07.

[19] "Fifty-move rule - Wikipedia." https://en.wikipedia.org/wiki/Fifty-move_rule. Accessed: 2022-06-07.

[20] "Manual for the `time` command on Ubuntu 22.04." http://manpages.ubuntu.com/manpages/jammy/en/man1/time.1.html. Accessed: 2022-06-15.

[21] "List of world records in chess." https://en.wikipedia.org/wiki/List_of_world_records_in_chess. Accessed: 2022-06-15.

[22] N. Wu, T. Schrijvers, and R. Hinze, "Effect handlers in scope," in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pp. 1–12, 2014.