Enhancing Functionality with Assistive Error Visualisations in Encore

Alexis Remmers
Abstract

Enhancing Functionality with Assistive Error Visualisations in Encore

Alexis Remmers

Compiler errors are an inevitable part of software development in compiled languages. Many compilers use very specific phrasing, which can be hard to understand by beginners, or programmers unfamiliar with constructs or concepts that are specific to a language. This is no less true for experimental research languages such as Encore, the language improved in this thesis. Encore introduces type system features seldom used in the mainstream.

Modern compilers output messages with a wide target audience, typically both human readers as well as tools that parse error messages and show them inline e.g., as part of an IDE tool chain. Balancing human readability and precision is a difficult tightrope walk.

This thesis reports on the design and implementation of an error handler for Encore for the displaying of error messages in a way that specifically targets human readers through systematic use of whitespace and color coding. The new handler also enables the compiler to provide further information, such as suggesting a probable problem fix. As a direct result of this work, the error messages in the Encore compiler have been improved considerably, and there is a clear path for integration of future error messages.
Acknowledgments

I would like to thank the people involved in the UPSCALE project and Encore for this opportunity and their help. Special thanks to Elias for the feedback and help in making sense of the code, as well as Alexander for keeping me company and motivating me to keep working.
Contents

1 Introduction 5
  1.1 Purpose and Goals ............................................. 5
  1.2 Motivation ..................................................... 5
  1.3 Methodology ................................................... 6
  1.4 Outline ......................................................... 6

2 Background 7
  2.1 Encore .......................................................... 7
  2.2 Abstract Syntax Trees .......................................... 8
  2.3 Encore Compiler Structure ..................................... 8
  2.4 The Production of Errors Today ............................... 9

3 Analysis 11
  3.1 Handling of Errors in Programming Languages and Tools .... 11
    3.1.1 Clang ...................................................... 11
    3.1.2 Elm ........................................................ 12
    3.1.3 Rust ...................................................... 13
  3.2 Available Information Throughout an Encore Compilation .... 14

4 Design 15
  4.1 Defining the Structure of an Error ............................ 15
  4.2 Tools for Consistent and Guiding Visualizations ............... 17

5 Results 18
  5.1 A New Era of Encore Errors ................................... 18
    5.1.1 Declaring and Describing Errors ......................... 19
    5.1.2 Positioning ................................................. 19
    5.1.3 Code Visualizer ........................................... 19
    5.1.4 Suggesting Fixes and Additional Notes .................. 21
    5.1.5 Assisted Enhancement with Error Explanation ............ 22
    5.1.6 Color Themes for Context ................................ 22
  5.2 Case: Create an Error in the Old System ..................... 23
  5.3 Case: Create an Error in the New System ..................... 25

6 Conclusion 26
  6.1 Future Work .................................................. 26

References 29
1 Introduction

Delving into a new programming language always entails some level of a learning curve; this also escalates if the sort of programming paradigm in the language deviates greatly from previous experience. But the compiler can help ease the understanding of new language nuances. Therefore, in the task of enabling a developer to recognize what needs to be done to solve a particular problem, the most important tool a compiler can contribute with is relaying an understandable error message.

But a compiler can be more pedagogical in its output. With a clear structure and consistent use of color and whitespace, the compiler can more easily direct the users attention toward key points and therefore more easily relay the context of an error. The compiler could also assist by giving hints on how to correct common mistakes.

1.1 Purpose and Goals

The purpose of this project is to improve how the Encore compiler [1] assists developers in understanding warnings and error messages. Specifically, the goal of this thesis is to study how programming languages, renowned for giving good feedback, structure their messages and, influenced by this study, change the existing Encore compiler infrastructure for error reporting. In particular, this thesis explores the possibility of integrating a simple analysis of compile-time errors and suggesting possible fixes during compilation.

1.2 Motivation

When a compilation fails, the compiler will output an error stating if a syntactical or semantical error has occurred and which file and line of code it occurred on, along with a description of the error and a printout of the faulty expression. This printout though is not a true representation of the actual source code, as illustrated below.

Figure 1 shows an instance of a type mismatch error, where the program tries to use the operation $+$ on a string which is only defined for numerical types. The error prints the expression in which the error was thrown from, as well as one expression above that since this was a nested expression. The output is neither the entire line which the developer has written nor is it formatted in the way that is stated in the source code in regards to spacing and parentheses. The reason for this behavior is explained in Section 2.4. By misrepresenting the code, the printout does not assist the developer in figuring out what is wrong or where the fault occurs. It could even be misinterpreted as the Encore compiler disregards parentheses and therefore could change the result of execution.
1 INTRODUCTION

If the Encore compiler would change how an error is rendered, so that it does not produce a misrepresented form of the source code, also by further assisting in any intricate concepts; the learning curve for Encore can be lowered for any developer interested to use the language.

1.3 Methodology

The initial step of this project is a survey of several existing languages’ handling and displaying of user errors as input for defining how the Encore compiler could be improved. It also involves an investigation on what information is available or can be made accessible during the different stages of compilation, to further enhance the output for an error. By analyzing how the compiler determines that an error has occurred and is produced, it is possible to provide a better understanding of how the compiler could provide further information and in a better visual manner. With this, a new design for error messages is made in conjunction with feedback from members of the Encore development team. With the new design and knowledge of how the Encore compiler operates, a new error handler is made which renders an error in a more assisted structure. The new handler is also designed to introduce the ability to give further aid with suggestions and supply additional information on the type of compilation error if needed. During the project, a discussion was made with the Encore team and it was decided that the visual representation of an error would follow the style of Rust language error message [2].

1.4 Outline

Section 2 covers some initial background of the Encore programming language and its compiler. Section 3 contains the analysis of how other modern programming languages handle and display errors, and what information can be obtained throughout the compi-
2 Background

This first section briefly covers the Encore language (Section 2.1). The following sections introduce the notion of Abstract Syntax Trees which is an intermediate representation of a program for most compilers (Section 2.2), the architecture of the Encore compiler (Section 2.3), and finally how the Encore compiler reports errors back to a programmer (Section 2.4).

2.1 Encore

Encore is a programming language in development at Uppsala University in Sweden, it is an object-oriented language based on the actor model [3]. While each actor is sequential, Encore uses a capability-based type system that statically guarantees data-race freedom when interacting between multiple actors concurrently [4]. This design allows the programmer to focus on the task at hand instead of managing thread synchronization and race conditions [5]. The Encore compiler is written in Haskell [6] and compiles Encore source code into C, which is then compiled and optimized using the established compiler Clang.

Encore supports both methods, bound by an object that can operate on that object’s fields and data, as well as functions, not bound by anything, that can only operate on its arguments. Encore requires that the target of a method invocation is explicitly given (e.g. this.foo() ). Similar to other object-oriented languages like Java and C++, Encore uses classes as blueprints for objects. Classes contain declarations of the methods and fields for its instantiations. Unlike Java and C++, Encore does not support inheritance between classes and instead uses traits to get the same benefit as inheritance or as Java interfaces. By sharing traits between different classes, the language can obtain polymorphism on objects abilities rather than class inheritance.
2 BACKGROUND

A. Remmers

Figure 2: Simple Abstract Syntax Tree

Figure 3: The Encore compiler passes. The output stage represents both the optimization as well as the code generation.

2.2 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is frequently used in compilers as an intermediate representation of a program [7]. When constructing an AST from a program, all expressions and operations are represented as nodes with its values or nested sub-expressions as subtrees as can be seen in Figure 2. If a program contains a syntactical error, the compiler cannot parse the code into an AST and produces a parse error (one example of this is if a binary operation that requires two operands but are only given one, such as "3 + ").

2.3 Encore Compiler Structure

When a program is compiled using the Encore compiler, the code goes through seven passes, each constructing an extended form of AST (visualized in Figure 3), containing first syntactical information, and later on semantical information of each node to determine syntactic or semantic errors. If a program is free from syntax or semantic errors, the last compiler pass translates the Encore source into the equivalent C code that can be linked with a modified Pony\(^1\) runtime library into an executable.

\(^1\)https://www.ponylang.io/
The Seven Passes of a Compilation

**Parsing** Constructing an AST from the source code and checking for syntax errors. This step is recursively done on all imported files and dependencies.

**Desugaring** Syntactic sugar are expressions that offer the same result as other operations but might have a higher level of abstraction or readability, this allows the language to offer more versatility for the programmer. The desugaring stage replaces the syntactic sugar the language allows, with the proper operands.

**Pre-checking** Resolves imports by transitively adding all imported modules to the AST and checks classes, methods and functions for namespace conflicts (e.g. two globally visible functions with the same name). The pre-checker also adds the type information on all declaration headers in the resulting AST.

**Type-checking** Checks the class declaration, fields, trait requirements, as well as the body of each method and function in the program for type conflicts and semantical errors. Resulting in the pre-checked AST, extended with type information.

**Capture-checking** Checks that the linear type system in Encore is upheld, meaning the linear references are either handled within a single thread or that the object is not accessed while being borrowed to another actor.

**Optimizer** Additional desugaring that were kept for type checking and changing some type-information to optimize the code generation.

**Code Generator** Generates the program in C as an intermediate representation and letting the Clang compiler [8] produce an executable by linking it with necessary runtime libraries.

2.4 The Production of Errors Today

There are four stages where an error in the source can be found during compilation as can be seen in Figure 3. During the parsing stage, if a syntax error is found a `ParseError`, will be produced. This kind of error is handled directly by the parser, the library megaparsec [9] and prints the state in which the error was caught and what the parser expected to find at that location. The other three stages; Pre-, Type- and Capture-checker produces a `TCError`, that hold the specific error that has been thrown, and a backtrace of the AST representing how and where the checker got to a non-valid expression. When an error is shown, it is displayed with three key parts: The line and column where the error occurs, a description on what kind of error was found, and a
**BACKGROUND**

```java
6 active class Main
7 8 def foo() : int
9 var str = "string"
10 11 (1 * str) / 3
12 13 1 * (str / 3)
14 end
15 end
```

---

*** Error during typechecking ***

"parens_order.enc" (line 11, column 12)
Operator '*' is only defined for numeric types
Left type: int
Right type: String.String
In expression:
1 * str
In expression:
1 * str / 3

"parens_order.enc" (line 13, column 18)
Operator '/' is only defined for numeric types
Left type: String.String
Right type: int
In expression:
str / 3
In expression:
1 * str / 3

---

**Figure 4:** The error output from two lines with different order of execution.

reduced version of the backtrace. If an error occurred during type-checking, the printed backtrace will only contain the two innermost expressions to not clutter the output.

For reasons of efficiency and good software engineering, many compilers utilise tricks under the hood to transform the users’ programs into an intermediate form that is eventually checked and—if correct—compiled. For example, when parsing parentheses that determine order of execution, the parentheses can be ignored by letting the ASTs internal representation handle the evaluation order. Another example is that many compilers internally only support a single loop construct, say while, and internally translates all other loops into while loops. This translation to the canonical internal form is handled in the desugaring phase. However, these tricks introduces problems for reporting errors back to programmers. When an expression is printed from the backtrace, they might reveal the desugared, internal structure of the program and no longer representing the source code, which is not very helpful for correcting errors.

In Figure 4 two lines with different order of execution are shown and a segment of their corresponding errors. By displaying two levels of nested expressions the compiler can imply it still conforms to the desired order of execution but does so in a way that could potentially confuse a programmer. The same circumstance applies to special characters such as newline and space; the parser ignores these characters and prints all variables and operands with a single space between them, disregarding how a programmer wrote the source code as was shown in Figure 1. By visualizing code segments in a representation that is not equivalent to the actual source code, it can obscure the location of the error and risk a developer trying to solve the issue at the wrong location.
3 Analysis

Section 2.4 gave an example of how Encore errors are rendered and how the compiler operates to explain why an error is rendered in the way it does. This section will examine some other compilers and tools, and describe how they inform programmers of errors (Section 3.1). And finally what information can be extracted during a compilation in Encore and how this could be useful to assist the development process (Section 3.2). With the information collected in this section, Section 4 defines a design structure for the error handler that is demonstrated in Section 5.

3.1 Handling of Errors in Programming Languages and Tools

Many compilers, especially older ones prioritized adding features or optimizing code generation rather than refactoring their codebase for better error reporting [10]. Instead, developers started to rely on external tools to assist them in finding bugs and displaying them in a understandable fashion, such as linters [11] and Integrated Development Environments (IDEs). Some of these external tools such as IDEs parse the compiler output for a better visualization inside their environment, further incentivizing compiler authors to keep their inherited legacy structure. There have been some projects, mostly for educational purposes, that built their systems to enhance the learning experience for beginners in a certain programming language. Two of these are Thesis, an ANSI C interpreter; and Helium, a Haskell compiler [12, 13]. Neither of these projects though are actively maintained.

There are some more modern programming languages and compilers that recognize the value of having more structured and visually guiding error messages that are in active development and highly utilized, such as Clang, Elm and Rust.

3.1.1 Clang

Clang\(^2\) is a compiler for C, C++ and Objective-C based programming languages utilizing the LLVM compiler infrastructure as its backend. When released, Clang was designed to be more user-friendly than its counterpart GCC. It does so by preserving source-level information which gets lost during parsing. For example, this allows the compiler to report correct line and column position for an error, even if preprocessed macros have moved the position. Clang was also designed to have human-readable diagnostics (error and warning messages) [8].

\(^2\)https://clang.llvm.org/
3 ANALYSIS

A. Remmers

Figure 5: Example of an error between GCC and Clang.

3.1.2 Elm

Elm is a functional language that compiles to JavaScript and is used to create websites and web applications. Elm is a statically typed language, using type inference for values and functions without concrete type-annotations. With this, it can detect edge cases and discrepancies and the Elm language advertises itself as with “practically no runtime errors”. If an error occurs the compiler shows a description of the error and a printout of the line along with a highlight of where the error occurred. Following the description

$ gcc-4.9 -fsyntax-only t.c

$t.c: In function 'int f(int, int)':
    return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);

$ clang -fsyntax-only t.c

$t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
    return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
    ^

Figure 6: Elm-error: Misspelled function.

Figure 7: Elm-error: Type with missing field.

Elm\textsuperscript{3} is a functional language that compiles to JavaScript and is used to create websites and web applications. Elm is a statically typed language, using type inference for values and functions without concrete type-annotations. With this, it can detect edge cases and discrepancies and the Elm language advertises itself as with “practically no runtime errors”. If an error occurs the compiler shows a description of the error and a printout of the line along with a highlight of where the error occurred. Following the description

\textsuperscript{3}http://elm-lang.org/
and source printout, the compiler tries to give helpful hints to help developers solve the problem [14]. In Figure 6, a function from the module List is misspelled, therefore the compiler gives some suggestions of possible alternatives from the module. In Figure 7, function isOver50 expects a record with the field age, but is later used with a record without this field, the compiler shows the two types, an arbitrary record with a comparable value age, and the record with its fields and subsequent types to help the developer to figure out what is missing.

3.1.3 Rust

```
error[E0186]: method 'foo' has a '&self' declaration in the trait, but not in the impl
-- src/test/compile-fail/E0186.rs:18:5
12 | fn foo(&self);    
  | ^^^^^^ expected '&self' in impl
error: aborting due to previous error
```

Figure 8: Rust-error: Argument mismatch between function declaration and function header.

```
error[E0412]: type name 'Mul' is undefined or not in scope
-- src/test/compile-fail/issue-21221-1.rs:72:16
72 | fn getMul -> Mul {      
  | ^^^^^ undefined or not in scope
= help: you can import several candidates into scope (`use ...;`):
= help: 'mul1::Mul'
= help: 'mul2::Mul'
= help: 'mul3::Mul'
= help: 'mul4::Mul'
= help: and 2 other candidates
```

Figure 9: Rust-error: Name not in scope, with suggestions.

Rust⁴ is a multi-paradigm system language that guarantees memory-safety and thread-safety [15]. They updated their error messages on what and how errors should be displayed and the possibility to provide helpful hints [2]. They also drew some inspiration from Elm to further extend their messages for more verbosity. Rust uses a defined structure of how an error should be represented: First by defining that an error has occurred with its following description, the file-path and position, a visualization of the code with highlighted code and hints embedded. In Figure 8 there is a mismatch between a function declaration and its actual implementation. The error shows both lines with different colors, blue as guiding information and red as the code that threw the error, with a short label with helpful hints. In Figure 9 there is too much information to fit in

⁴https://www.rust-lang.org/
A short label, therefore it gets attached as an additional note. In both Figures, there is also an error code. This can be used to get an extended error message on the type of error that occurred, what the usual reasons are and how to solve them. This has been deemed a great way to help bridge between an error and learning unfamiliar concepts in the Rust language.

3.2 Available Information Throughout an Encore Compilation

When a semantical error occurs in Encore, the compiler has some understanding of the context in which the faulty code appears. This context is called the Environment in Encore and is a table containing information of all visible functions, methods and local variables throughout the compilation. Figure 10 shows an illustration on what data is held inside the Environment and a subset of its nested structures. Broadly speaking, the Environment holds the method- and function-names along with its arguments and type information, for every class and trait, for every imported module. It also contains any local information needed throughout the compilation such as local variables, their type declarations and a backtrace of the compilers evaluation.

As of now, when an error occurs, some key information gets taken out of the environment to contextualize the description of an error (for example: including the type name in a type mismatch or method name in an insufficient arguments error). If the Environment in its entirety would be passed alongside an error, the compiler can, based on the type of error and its inherent complexity, analyze the code to give suggestions for a possible solution when printing the error.
For example, as stated in Section 2.1, Encore supports both methods and functions; a call to a non-existing function can be simply a spelling error, or the name corresponds to a valid method bound to the object instead of a function, but accidentally forgetting to explicitly name the receiver (e.g. `foo()` as opposed to `this.foo()`). With the Environment, the compiler could then compare the undefined function name against all visible methods and functions, if the match is within a certain threshold, the compiler could suggest that as a possible solution. Another example would be if a developer attempts to call a global function (i.e. not a lambda function assigned to a field of an object) but still prepends `this` on the function call. This syntactic form will be interpreted as a method call, and if there is no method in the given object with that name, it would run into a `MethodNotFoundError`. With the Environment, it can check if there is a visible function with the same name and if there is, suggest that if the intention was to use a function, to remove the prepended object-reference.

4 Design

The previous section showed how other compilers design and handle their errors and what kind of information could be extracted from the Encore compiler when an error has occurred. In this section we introduce the new design for error messages in Encore, inspired by the languages covered in Section 3.1, and evaluate different tools needed to achieve the defined structure and design (Section 4.2).

4.1 Defining the Structure of an Error

In collaboration with the Encore team during the project, it was decided that an Encore error should follow the style of a Rust compiler error [2]. Figure 11 breaks down the components of a Rust error message, which we based the Encore error message design on.

![Figure 11: A Rust-error with annotations on the separate segments an error is constructed by.](image-url)
The New Encore Error Design Structure

- Declare that an error occurred and possibly what type of error.
- Give a short description of the error.
- Display the file-path and position of the fault.
- Show the location of the error in the source code and possibly some surrounding lines for context.
- Optionally provide a short suggestion or note inlined with the source code at the errors location.
- Optionally provide a longer note underneath, but still visually connected to the error to maintain context, if the type of error demands further explanation or if the suggestion/note is too large to fit in-lined into the error.

The error uses white space and color to highlight what information is crucial to understanding the error. Research has found that by using colors in key points of information, readers are faster on extracting the necessary information needed [16], also that by structuring data with linebreaks and whitespace the readability is elevated [17]. By intensifying the color of the description (by using a brighter color and/or bold font depending on terminal support), the error tries to relay that the information is crucial to understanding what went wrong. By using the same color on the error declaration and the highlight on the code, it relays that these two are related as can be seen in Figure 11. By using characters of a separate darker color (i.e. the dark blue characters) to act as a scaffold, filling the whitespace, it provides a contextual frame, enclosing what relates to that specific error.

Color will be used systematically according to the following scheme:

**Classification** To clarify what type of output will be presented (e.g. error or warning).

**Information** Visualize key points of information an error wants to relay, as the description or keywords in notes or suggestions.

**Data** Information related to finding the faulty code such as the position and the visualized source code. To abide with different developers preferences this could be determined by the developers color palette but should still be distinct from the **Information** colorization.
Highlight If anything shown needs to be highlighted so that the developer gets notified of a specific location or information, e.g. the position in which an error occurred, this should be related to the **Classification** colorization to maintain context.

Logistic Non-detrimental information to maintain the contextual frame of the error message and highlight different parts of the error (e.g. in Figure 11 an arrow is used to highlight the position, and a numbered blue border is used to highlight the source code).

### 4.2 Tools for Consistent and Guiding Visualizations

Encore uses a library for formatting text and source code in a stylistic convention (called prettyprinters) named Pretty [18]. The compiler did not have a way to perform colorized output so an investigation was made on how to extend the functionality of the compiler to add the ability of colorized output, either by replacing the included prettyprinter (pretty-1.1.2.0) or extending the functionality with an additional library. At the same time without adding an abundance of dependencies for a trivial task, or be overly complicated which could deteriorate the maintainability of the error-handler. The options investigated were:

**Rainbox** Defines colorized two-dimensional text-boxes which can be stitched together in a grid-like fashion [19]. Determined to be overly complicated for the task and would be hard to maintain in future development.

**Rainbow** Core color component of Rainbox [20]; can output 8- or 256-colored text on supported terminals. This library turned out to be hard to incorporate into the rest of the compiler and would only work on Unix-like systems.

**Ansi-terminal** Providing support for ANSI control character sequences for terminals on Windows and Unix-like operating systems [21]. This package enters the escape code directly in the terminal and expects the coming text to directly be printed. This made it hard to modularize the printer and maintain consistency if multiple colors were needed on a single line (as the example of printing a single or multiple lines in Figures 16 & 17).

**Prettyprinter** A modern pretty printer with an extension for colorized output [22]. Conflicted with *Pretty* that was included previously in Encore. After replacing the packages it was discovered the two libraries defined an empty document differently, breaking previous functionality. In Encore you can pretty-print the source code giving you the same code back but adhered to Encore coding convention for indentation, linebreaks, and structure. This results in either having two very equal libraries as a dependency, or a major refactoring necessary to uphold functionality.
Pretty Version 1.1.3.1 and later introduced an annotated form of the document data type, making it possible to define how to render each annotated segment separately. This made it possible to construct a renderer using ansi-terminal. The issue with this was that the Haskell package resolver used in the Encore project did not have version 1.1.3.1 and thus needed to either include an unsupported package or update the resolver with the possibility of other packages breaking.

The Encore team updated the resolver so the updated version of Pretty could be used. With this, each segment of a text document can be extended with annotations containing information related to how it should be represented. This results in that details on how a specific part of an error should look or print can be abstracted away from further development. By letting the error handler encapsulate each part of an error with annotations based on the color classifications from Section 4.1, the handler can then ensure that whatever text an error produces, the visualizations will be consistent throughout every error.

5 Results

After constructing a design guide in Section 4.1 as well as determining the tools to achieve it, this Section will demonstrate the result of the implementation, and how it maintains its design.

5.1 A New Era of Encore Errors

The new handler was rewritten to be able to reuse as much of the descriptions of the old version as possible so that it does not need a complete rewrite for all errors to be a viable replacement. Each component of a rendered error, highlighted in Figure 15, is generated by a separate function so its able to be individually maintained, or customized for a given error type if it can further assist a developer, a subset of these differences can be seen in Figures [12–14]. The following subsections will describe how the new

---

**Figure 12:** Comparison between the old and new error handler with the source given in Figure 1.
handler adheres to the design structure from Section 4.1 and additional features such as the ability the construct verbose explanations for a given error type.

5.1.1 Declaring and Describing Errors

When displaying the declaration, the handler can include an abstract name or error code of the specific type of error, this is to have the ability for a more in-depth explanation for the error in question (see Section 5.1.5). If no such explanation exists, it will display that an arbitrary error occurred. The description is a short summary of what has gone wrong and what the error is about. By reusing the same descriptions as the old handler, all existing errors are compatible with the new handler without additional work.

5.1.2 Positioning

One of the most important aspects of an error message, apart from stating what error occurred, is its location in the source. The handler will use the symbol `-->` to show a developer that this is the position where an error occurred, then stating the file path, line and column in which the error started. The position is always placed on a separate line underneath the declaration and description to add verbosity for new developers in the Encore [1] language while being as succinct as possible to have the ability to be quickly skimmed over by more experienced users.

5.1.3 Code Visualizer

When rendering and highlighting source code, the new handler is built to be as adaptable to the developers’ code as possible. It shows the source code that is written as the developer wrote it, so that it does not differ between the error message and the actual code, included in the code is a highlight on what particular part of the source code threw the error. Depending on how the faulty source code is structured, the handler changes its representation of a highlight to be as readable as possible: If the error occurs on an expression that is only one line high, it will display only that line with a highlight underneath to visualize what part of the code that is faulty (Figure 16). If the error occurs spanning multiple lines (e.g. a function call where a developer separated the arguments with new lines, and therefore the position ranges over the entire function call), the visualizer has to handle that as well. This will highlight the start- and end-position of the error and encapsulate the entire range with lines as can be seen in Figure 17.
**RESULTS**

A. Remmers

---

**Figure 13:** Option to not display the source code even if a position has been given, the reason in this instance is due to future work (Section 6.1).

---

**Figure 14:** With no rewrite of the description it will try and break into new lines at better optimized positions while the old will wrap around a the terminal width.

---

**Figure 15:** Each colorized part is a separate function able to be individually modified, or removed if the error type requires it.

---

**Figure 16:** New Encore error for argument arity mismatch.
5.1.4 Suggesting Fixes and Additional Notes

By exposing the environment (Section 3.2) data structure to the error handler, the compiler can, based on the type of error, try and analyze the faulty code and give suggestions or notes for a possible solution. Figure 18 shows an error where the program calls for a non-existing method, but there is a function with the same name and arguments and therefore hint the programmer towards the function. The handler has two ways to make suggestions; either as a short text in-lined with highlighting the source or as a longer segment underneath. Utilizing the suggestions is entirely optional for the Encore compiler developers, some errors are problematic to give a good solution for, or might be too computationally complex to validate the compilation holdup. For the developers of Encore, there are no constraints on which of the locations are to be used for suggestions, in Figure 17 the handler uses both, the short suggestion on what the function expected, and the long suggestion as a more detailed note on what was expected versus what was found. Due to the fact though that the smaller suggestion will be in-lined with the source code, a recommendation is to only provide the short suggestion if it can be determined that it will produce a short output.
5.1.5 Assisted Enhancement with Error Explanation

There is a wide range of unique errors that can occur in a program, especially since some programming languages have subtle features that are unusual among other languages. For example, Encore’s linear type system is unusual with respect to most mainstream programming languages and might be difficult to understand for a programmer who has never worked with such a system before. Therefore when given an error, a code for the error type could be given in which the user could directly inquire the compiler for a more comprehensive documentation on what this error entails and a short example. This gives a developer more power and ability to understand better the more complex nuances of the Encore programming language. For example, if an error produced the error code E0073, that code could be passed to the compiler with an “–explain” flag resulting in the explanation as seen in Listing 1

```
This error occurs when the compiler was unable to infer the concrete type of a variable. It can occur for several cases, the most common of which is a mismatch in the expected type that the compiler inferred for a variable’s initializing expression, and the actual type explicitly assigned to the variable.

For example:
```

```
let x : int = "I am not a number!"
//   ---   ---------------
//    |     |
//    | initializing expression;
//    | compiler infers type `String`
//    | type `int` assigned to variable `x`
```

Listing 1: Example of a type mismatch explanation.5

5.1.6 Color Themes for Context

With the updated library Pretty mentioned in Section 4.2, each part of the handler can be individually annotated a color classification (defined in Section 4.1), detailing the information related to how it should be represented. But the annotation does not define what colors are to be outputted. When rendering, depending on if it is an error or warning, the handler will use different color themes to determine what color is to be displayed for each annotation, illustrated in Figure 19. This results in that the

5Text is from Rust: rustc –explain E0308, the code is changed to Encore syntax
Warning at "warnAndError.enc" (line 6, column 9):
Result of '34' is discarded

*** Error during typechecking ***
"warnAndError.enc" (line 7, column 9)
Expected String.String (type of the enclosing method or function) but found expression of type 'int'
In expression:
  return(3)
In expression:
  do
    34
  return(3)
end
In method 'main' of type 'String.String'
In class 'Main'

Warning: Result of '34' is discarded
--- "warnAndError.enc" (Line:6, Column:9)
| 6 | 34
| ^^

*** Error during typechecking ***
Error: Expected String.String (type of the enclosing method or function) but found expression of type 'int'
--- "warnAndError.enc" (Line:7, Column:9)
| 7 | return 3
| ^^^^^^^

Aborting due to previous error

Figure 19: A comparison of a warning between the old and new handler.

handler does not have to define the color of each particular part, only define what kind of information it contains and the renderer will use the correct colors. This modularity would be more evident if the handler would be extended to handle more types other than semantical errors and warnings. For example, if the new handler would also visualize ParseError (Section 2.4), they could get a separate color theme with no additional changes made to how a position is constructed, or how the source code is visualized. The renderer disables all colorizations if it detects that it will not output into a terminal, this is due to that the colors are encoded with ANSI escape codes telling a terminal to change the color. By disabling colorization if the output is redirected into a file or piped to another program, it will not capture the escape codes and only receive the clean output expected.

5.2 Case: Create an Error in the Old System

If a developer would like to enter a new error to the Encore compiler using the old system, they would first have to define it. The error definition has to contain all information that would be interesting to be stated in the error output. For example: The error BinaryOperandMismatchError is thrown when a binary operation is used with types not defined for that operator (i.e. operator & is only defined for boolean types), therefore the definition is stated as BinaryOperandMismatchError BinaryOp String Type Type so that when the error is shown, it includes the operation (BinaryOp), the type name of which the operation is defined for (String), and the types of the left- and right-hand side expressions (Type). When displaying an error the compiler will use the string representation of said error, using Haskell’s show operator as can be seen in Figure 20.

With the old handler, the description of an error is the only way to define how an error is supposed to be displayed, and what information could be useful to assist the programmer. Therefore all information that could be useful to be displayed, has to be included in the definition of said error, seen in Figure 21.
show (BinaryOperandMismatchError op kind lType rType) =
printf ("Operator '%s' is only defined for %s types\n" ++
  " Left type: %s\n" ++
  " Right type: %s")
(show op) kind (show lType) (show rType)

Figure 20: Actual implementation of an BinaryOperandMismatchError

*** Error during typechecking ***
"binarymismatch.enc" (line 7, column 17)
Operator 'and' is only defined for boolean types
  Left type: bool
  Right type: int
In expression:
  true && notBoolean
In expression:
  if true && notBoolean then
    return(0)
  else
    return(1)
end
In method 'baz' of type 'int'
In class 'Foo'

Figure 21: Output of a BinaryOperandMismatchError. Note that the line-
breaks and indentation in the description is manually inserted in Figure 20.

*** Error during typechecking ***
"binarymismatch.enc" (line 8, column 17)
Operator 'and' is only defined for boolean types
  --> "binarymismatch.enc" (Line:8, Column:17)
  if true && notBoolean then
    **
  else
    Aborting due to previous error

Figure 22: The left shows a BinaryOperandMismatchError with no addi-
tional changes. The right shows the same error but utilizing the long sugges-
tion as a note.
5.3 Case: Create an Error in the New System

The minimum number of work for adding new error messages to the Encore compiler with the new handler is on par with the old handler. The error message will be displayed as its description and the source will be visualized with no additional changes, but now all information does not have to be included in the description. When determining that an error should be thrown they could still include all information that determined the error as its definition (e.g. the type information of the left- and right-hand side expressions in the BinaryOperandMismatchError) as this information is already evaluated. As a note, all information is not saved into the Environment (Section 3.2) before an error is thrown, and if that information would be beneficial when making suggestions, it could be difficult to re-evaluate the information, while some information will be included in the environments locals-table, it’s easier to retrieve the specific information if it is included in the definition.

Information that could be valid to present to a programmer should either be stated in the error description, as a quick hint in the short suggestion, or as a longer note underneath the error as a longer suggestion. An example of this can be seen in Figure 22 where some of the information previously in the description has been moved to a long suggest. When defining an error there is the possibility that the Encore developer know some probable causes when said error will be thrown, for example: Unknown variable names could be a spelling error, undefined function calls could be known but inside a module that is not imported; features unique to the Encore language could be forgotten during coding. Now, the developer can construct a function to determine if any of these cases are true by querying the Environment and return with a possible solution. The resulting information should be included in either the short or long suggestion.
Last, to provide further assistance, an explanation of the error can be provided. The extended explanation is provided in a separate text file that should explain a little about what the specific error means, what is a probable cause with a possible solution. An example of such an explanation is shown in Figure 1 on Page 22. These explanations are given a unique ID based on the error which will correspond to the error code that will be displayed and have to correspond to the filename of the explanation.

6 Conclusion

This thesis presents a redesign of the Encore error handler to further assist programmers by producing more readable and understanding errors, and by adding the possibility to let the compiler analyze the source and suggest solutions to said error, while at the same time make it easier for further development. The restructuring of Encores new error handler was heavily influenced by the design of error messages in Rust. The previous error handler in Encore already had a short description for every error type and could relatively easily be turned into descriptions in a similar design as Rust. The new design structure is also more defined; with the use of hash-codes for errors to obtain more information, it provides the verbosity and assistance for beginners of a language, without being in the way for more experienced developers.

The new error handler provides more pedagogical guidance on what has gone wrong and where for each error. By using different colors on key parts, users are quicker to find the relevant information [16], also by adding whitespace between information enhances readability [17, 23]. The new handler is also built more modular so that developers of the Encore compiler can easily extend or customize the output for a specific error, either by adding suggestions or notes with further information or changing parts of the visualizer.

6.1 Future Work

As stated in section 2.4, the previous method of printing an expression or code snippet did not produce a result that was the same as the written source code. Therefore, the choice was made to read the file once more to get the source code printed as the developer wrote it. To solve this the compiler could either let the parser include metadata such as whitespace, line-breaks, and parentheses into its AST and the printer could optionally include it. Another way of handling it is if the source code of all included files could be passed along throughout the compilation, the error handler would then have the information needed without additional I/O (but at the cost of more memory use throughout the compilation). Either of these solutions would aid in the possibility of printing multiple locations in the source files so that the handler does not have to read
Error: Left-hand side of operand is not assignable
---> "plusequals.enc" (Line:32, Column:37)
| 32 | println("x = ", (e*2) += 3)
| ^^^^^^ Can only be used on var or fields

Figure 23: The operation "+=" is only two characters long, but its positional range encompasses the four spaces afterwards as well.

Error[E0073]: Type 'real' does not match expected type 'int'
---> "methodArg.enc" (Line:4, Column:5)
| 4 | def foo(i1 : int, i2 : int, i3 : int = 5.3) : unit
| 5 | |
| 6 | end
| = note: expected type int
| found type real

Error[E0073]: Default type 'real' does not match expected type 'int'
---> "methodArg.enc" (Line:4, Column:33)
| 4 | def foo(i1 : int, i2 : int, i3 : int = 5.3) : unit
| ^^^^^^^^^^^^^

Figure 24: Type mismatch in method declaration, where the third argument is declared as int but is given a default value of type float. The original to the left will display the entire method head and body, while the concept to the right will only highlight the mismatching parameter.

all locations in advance. One example of this is if the arguments of a function or method call are wrong, the handler would more easily be able to print the actual declaration to compare.

With the new handler highlighting the faulty code some visual discrepancies have been found. Whitespace is not necessary to determine execution and is therefore ignored by the parser, the same rule applies to comments as well. Every token in the Encore language has a positional range where it begins and ends, but the determination of end positions gets evaluated after the parser has ignored the trailing whitespace and comments making the range of a token not representable to the actual size of the token. This results in that when an error occurs and the handler prints out the code, its highlighting will extend after the faulty operation until the next token, see Figure 23.

Some errors thrown by the type-checker could be specialized further to better represent and visualize individual errors. For example; Figure 17 on Page 21, shows a type mismatch on the fourth argument, the function expects a string but is given an integer. The error thrown gave the position of the entire method call instead of the actual argument, making it highlight the call in its entirety. This is also shown in Figure 24 where the fault is in the actual declaration, and as a cascading effect, its entire body be included when visualizing the error. These errors could be better produced if the error type would be more specified. Either by including the index of the argument, in which the handler could figure out the position to highlight, or have the specific argument as the position for the error. Both of these solutions would allow the handler to show the entire multi-lined method call, but only highlighting the actual argument making it more clear what the error is about, and specifically where it occurred.
Since the new handler reuses the description from the old error messages but with extended ability to show information, some of the error types could use the additional positions such as the long suggestions to note some of the information not necessarily needed in the description. Beyond that, as there are only a few kinds of error that are analyzed for the moment, there is a need to further expand these options to more errors and especially to the kinds that revolve around unique and unusual features of the Encore language. This need for further development also befalls the explanations described in Section 5.1.5. There are a few explanations included, and these can be used as references for how an explanation should be constructed.

In addition to the above ones, some minor improvements have been left out mostly due to time constraints that affect the result:

- If an error occurs on a position that ranges over an excessive amount of lines such as the instances mentioned previously when an entire function body will be displayed, the handler should truncate the result so that it does not clutter the output while at the same time maintains the context.

- To give a consistent and uniform visualization for all errors and warnings, the new handler could be extended to also be applied to parse errors. Parse errors are for now printed by the megaparsec library [9] and does not have the same structure as the other compilation errors. But it can provide the necessary information needed to extend the Encore error handler. A concept of this can be seen in Figure 25.

- The production of suggestions could be optimized further, the two suggestions are assigned by two separate functions. If they would be constructed by a single function, not only could calculations and data be reused, but the function could have the ability to dynamically determine where and what the information should reside, depending on what solution it could arrive to and how long the output would become.
References


