GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair

Francisco Ribeiro
francisco.j.ribeiro@inesctec.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

José Nuno Castro de Macedo
jose.n.macedo@inesctec.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

Kanae Tsushima
k_tsushima@nii.ac.jp
National Institute of Informatics
Sokendai University
Tokyo, Japan

Rui Abreu
rui@computer.org
INESC-ID
University of Porto
Porto, Portugal

João Saraiva
saraiva@di.uminho.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

Abstract

Type systems are responsible for assigning types to terms in programs. That way, they enforce the actions that can be taken and can, consequently, detect type errors during compilation. However, while they are able to flag the existence of an error, they often fail to pinpoint its cause or provide a helpful error message. Thus, without adequate support, debugging this kind of errors can take a considerable amount of effort. Recently, neural network models have been developed that are able to understand programming languages and perform several downstream tasks. We argue that type error debugging can be enhanced by taking advantage of this deeper understanding of the language's structure. In this paper, we present a technique that leverages GPT-3's capabilities to automatically fix type errors in *OCaml* programs. We perform multiple source code analysis tasks to produce useful prompts that are then provided to GPT-3 to generate potential patches. Our publicly available tool, MENTAT, supports multiple modes and was validated on an existing public dataset with thousands of *OCaml* programs. We automatically validate successful repairs by using *Quickcheck* to verify which generated patches produce the same output as the user-intended fixed version, achieving a 39% repair rate. In a comparative study, MENTAT outperformed two other techniques in automatically fixing ill-typed *OCaml* programs.

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623522

CCS Concepts: • **Software and its engineering → Software evolution**; *Software creation and management*; Software post-development issues.

Keywords: Automated Program Repair, GPT-3, Fault Localization, Code Generation

ACM Reference Format:
Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623522

1 Introduction

Programming languages usually have an associated type system responsible for determining whether some operation can be applied to some program term. This system ensures a program's correctness in terms of type safety. That is, if a program does not *typecheck*, it signals a logical error related to the inherent type constraints. However, even after knowing there is some type inconsistency, we still need to understand where and why that error occurred. In other words, a type system may be unable to provide the location of the error and the explanation as to why the error arose.

Undeniably, programmers are not completely left in the dark in this regard. Several programming languages provide *type inference* systems, which compute the expected type of expressions in the code. Despite considerable effort [9, 21, 24, 43, 47, 49] to provide helpful information for type error detection, compilers often fail to pinpoint the true cause of an error. Consider the following ill-typed *OCaml* program:

```
let rec add_list lst = match lst with
  | [] -> []
  | fst :: rest -> fst + (add_list rest)
```

Program 1. Ill-typed function: patterns differ on returned types

Program 1 consists of a recursive function `add_list` that takes a list of integers and should calculate the sum of all numbers. `ocamlc` would yield the following message[1]:

```
3 |        | fst :: rest -> fst + (add_list rest)
                             ^^^^^^^^^^^^^^^^
Error: This expression has type 'a list
       but an expression was expected of type int
```

The type system successfully detects a type error in the program and the compiler provides a message reporting the problem. If we replace the use of the *plus* operator (+) in line 3 by the *cons* operator (`::`), the whole program is well-typed. However, the expression highlighted as being problematic by the compiler is not the true origin of the error. As a consequence, the information about the mismatch of the expected type (`int`) and the inferred type (`'a list`) does not provide meaningful advice into how to approach the problem. Another way to fix this program, which corresponds to the programmer's intended fix, is to have it return zero (0) instead of the empty list (`[]`) in line 2. Hence, it often happens that the user's intended modification differs from what the compiler points out. That is because reporting type inconsistencies is influenced by the order in which expressions in a program show up. As long as no inconsistencies are detected, the inferred type for an expression is considered to be correct. As a result, the type system will have a *left-to-right* bias and errors tend to show up towards the end of a program [20]. Now consider that we swap the two patterns:

```
let rec add_list lst = match lst with
  | fst :: rest -> fst + (add_list rest)
  | [] -> []
```

This time, we get a different error message:

```
3 |        | [] -> []
                  ^^
Error: This expression has type 'a list
       but an expression was expected of type int
```

This means that the type error we are dealing with can have multiple causes. Depending on the order of the patterns, the cause that is reported changes.

However, even after recognizing the inherent limitations of type systems in accurately locating and explaining type inconsistencies, we are still left with fixing them. Automated program repair (APR) aims to generate patches for incorrect programs (either syntactically or semantically) with minimal human intervention [18]. Many approaches have emerged based on the competent programmer hypothesis or, put in other words, programmers "create programs that are close to being correct!" [13]. We argue that automatically finding repairs that eliminate type inconsistencies is one effective way of locating and understanding the root of a type error.

In this paper, we present an approach that leverages the code understanding and generation capabilities of models based on GPT-3 to automatically fix type errors in *OCaml* programs. Our focus is on analyzing the source code of ill-typed programs and generating prompts that are then provided to the model. By doing this, we aim to produce programs that are free from type errors and, thus, can be used to find and understand what was causing them. Our contributions are:

1. a source code analysis and manipulation technique that produces different kinds of prompts intended for GPT-3-based models (Section 3);
2. a publicly available tool, named Mentat, implementing this technique (Section 4);
3. an initial validation on a small set of programs, followed by a large-scale evaluation on an independent repository, with an analysis of the results obtained (Section 5);
4. a comparative study between our tool, Mentat, and two other techniques, namely Rite [41] and Seminal [25] on a common dataset, alongside the obtained insights.

Even though this work is concerned with type error debugging, it differs from previous approaches, which aim to improve the quality of type error messages [12, 24, 51], provide interactive type debugging [6, 7, 9, 48], and narrow down the area for type error debugging [19, 37, 42–44]. Instead, our work focuses on the automatic repair of type errors. We achieve this by analyzing and transforming source code and outputting it in a form that can be understood and processed by GPT-3, a large language model trained by *OpenAI*.

For the initial validation, we find that our tool presents at least one valid solution for each test program, with the *Fill* operation mode obtaining success rates varying from 53% to 60% for simple programs and from 83% to 100% for Dijkstra algorithm implementations. Regarding the large scale evaluation, we analyzed 1,318 buggy programs and were able to fix 516 of them, reaching a 39% repair rate. To automate this process we used two key features of property-based testing [10]: firstly, we automatically generate a very large number of random inputs, and secondly we define a property that tests whether the user-fixed program outputs the same result as the automatically repaired one. The program-specific property is also automatically generated, thus having a fully automated large scale validation process without relying on human intervention to inspect the generated patches. Also, we showed the potential for partial fixes by considering the results for programs that do not pass 100% of test cases. While the other operation modes perform worse overall when compared to *Fill*, they are still capable of generating successful results and, in some cases, succeed where *Fill* fails. Moreover, we performed a comparative study of our technique with two type repairing approaches, namely Rite [41] and Seminal [25]. Our first results show that Mentat gives the best program repair results with a 37.5% repair rate versus 33.4% from Rite and 7.8% from Seminal.

2 Background

A *type system* is a set of rules governing how data is represented and used in a program [34]. It is lightweight and does

[1] In *OCaml*, polymorphic type names are prefixed with a backquote.

not require special knowledge from the user. *Type inference* is a static algorithm to find the type of each part of a program without the programmer's annotation. The advantages of type systems include not only type-safety of programs but also efficient computation by enabling the generation of optimized code. For this reason, *Ruby*, a dynamically typed language, has recently introduced type inference, and *Python* has also introduced type-related features.

Originally, research in natural language processing (NLP) focused on ways of processing, analyzing, and manipulating natural language through statistical and rule-based modeling. More recently, the use of artificial intelligence has allowed the development of techniques that make NLP one of the most prominent fields in computer science. Simply put, NLP is responsible for encoding text into more appropriate machine level representations and also for processing and transforming these lower level descriptions into other forms of text. More specifically, neural networks have been very impactful in the development of NLP models. Some of the most important ones are BERT [14] and GPT [36] which have seen their utility displayed in an overwhelming amount of more specific downstream tasks. Encouraged by the achievements conducted in this area, the software engineering community has successfully applied some of NLP's fundamentals to build tools that improve software development workflow. Code completion is one of the most popular features and many code editors implement it one way or another. With the intent of going beyond basic level completions like more pertinent suggestions for API calls for a given context, the research community has also directed its efforts into developing versions of the BERT and GPT models that are specifically tailored towards programming languages. New models, such as CodeBERT [17] and CodeGPT [27], were created based on the original architectures. GPT-based code models are able to generate long and relatively complex code sequences by analyzing and inferring the context of the source code provided as input. One of the most recent iterations of such models is GPT-3 [4], which presents a high degree of success when employed in different scenarios such as cloze and completion tasks.

3 Technique

With our contribution, we intend to, for a given faulty *OCaml* program, extract as much information from it as possible, and then format it in a way that GPT-3 can understand and process it. For this, we first check for type inconsistencies. If they are present, we employ three different tasks: *type error location*, *inlining*, and *type unification*, with each being described in detail in the following sections. Figure 1 illustrates how the tasks generally interconnect and highlights them with the corresponding label. Nodes with dashed borders represent steps in which we make use of existing components and are not directly part of our contribution. Grey nodes and white nodes represent elements and actions, respectively. Depending on the way we wish to interact with GPT-3, the tasks may be combined in slightly different ways.

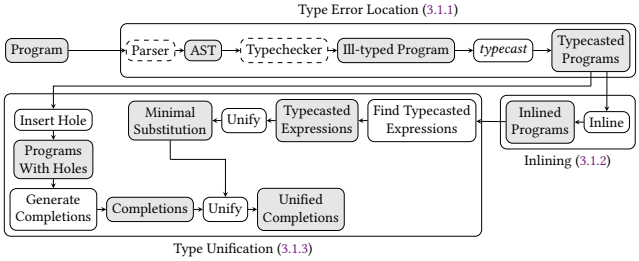

Figure 1. Interconnection of source code manipulation tasks

3.1 Source Code Analysis

3.1.1 Type Error Location. The compilers of strongly typed programming languages tend to check for source code errors in two separate steps when building an executable file: parsing and type checking. The parser checks whether the input is syntactically correct and if so it produces an *AST (Abstract Syntax Tree)*. The type checker traverses such a tree to check whether the underlying program obeys the type rules. If it does parse but it does not typecheck, then there is a type error, which is the focus of this work. If it does both parse and typecheck then, for our purposes, the program is considered correct.

Some programming languages offer us type conversion and manipulation tools; we focus on type conversion tools from strongly typed languages. Let us consider a function from now on referred to as *typecast*, which forcefully converts a value of any type into a value of any type. Of course, such function will not be able to actually do these conversions during a program's runtime, but it will be able to trick the compiler into interpreting an expression of one type as if it had a different type. As such, the *typecast* function will only be used when typechecking a program. In *OCaml*, this operation can be performed by using `Obj.magic`[2], which we will use in our tool. Any program referred to as *typecasted* from here onwards is a program in which part of it was transformed with the *typecast* function.

Recall the introductory example in Program 1. Because any type can be converted into any type with this function, we can, for example, apply *typecast* to the empty list (`[]`) to transform it into a different type *'b*, which the typechecker will deduce to be *int*. We could also apply *typecast* to the *plus* operator (`+`) thus transforming it into a function of type *'b* which the typechecker will deduce to be *'a* → *'a list* → *'a list*. Finally, we can also *typecast* the expression on the right-hand side and have the typechecker infer the type *'a list*. After parsing the *OCaml* program, we create multiple program

[2] `Obj.magic` has the type *'a* → *'b*

variants, each with the *typecast* function applied to a single expression, and then type check each application. Every variation of the original program that typechecks correctly is stored; if changing the type of one value / expression with *typecast* fixes the program type-wise, then we consider that the replaced value can be the error that needs to be fixed. Finally, we replace the usage of this function with a mask, signaling a hole in the program that needs to be filled. The following tasks will focus on analysing and transforming the program variations (which we call the *typecasted* programs) produced in this task.

In this work, the focus is on type errors with a single location. Nonetheless, our approach is still flexible to some instances of errors with multiple locations if all of them are contained within a single function call expression.

3.1.2 Inlining. We make use of *inlining* not for the usual purposes of compiler optimisations, but to be able to make information available from some parts of the program in other places. That is, the *inlining* step we describe here is done on the actual source code to improve its analysis further ahead, and not to produce more efficient machine code. This step is particularly useful as it allows us to extract better results from later type unification and type inference tasks. Consider the following *typecasted* program:

```
let f x = (Obj.magic (&&)) x (x + x)
```

Program 2. Obj.magic hides the error from the type system

If we ask for the type of the `Obj.magic (&&)` expression, the type system will infer it to be $int \rightarrow int \rightarrow 'a$. However, let us now extend the program with a test case:

```
let f x = (Obj.magic (&&)) x (x + x)
let t = (f 1) = 3
```

The second line specifies the usage of function `f`. By inlining function `f`, we associate it to a context in which the type system can take advantage of the extra information provided by the *int* parameter. As a result, the inferred type of the *typecasted* expression would be $int \rightarrow int \rightarrow int$.

To accomplish this *inlining* step, an environment is maintained throughout the underlying *AST* traversal. When a new definition is found, its identifier is stored and associated to the corresponding expression. As such, when the usage of an element stored in the environment is detected, the usage of the identifier is replaced by the expression's body, effectively *inlining* that piece of code. Special care needs to be taken for two scenarios: recursion and function arguments. For the first one, we need to avoid repeatedly *inlining* the same element as that could potentially lead to a non-terminating procedure. Nonetheless, there is still interest in performing this step once for recursive definitions. Thus, we allow *inlining* to happen exactly once in such cases. For the second scenario, most programming languages allow re-definition of variables in different scope levels and *OCaml* is no exception. It is possible to have a variable `x` already defined, and still define a new `x` in an inner scope. When inlining, in this case, we take care to inline the correct definition for the correct `x` variable. The inlined source code is only stored in memory and the original program is not modified, with GPT-3 never seeing the inlined version.

3.1.3 Type Unification. We make use of type unification to filter elements from a list of completion suggestions. Almost every code editor provides the ability to have completion suggestions on request from the user by specifying a place in the source code. Prior to requesting a list of completion suggestions, we replace *typecasted* expressions (obtained as described in Section 3.1.1) with typed holes. Then, we make use of an *OCaml language server (LSP)*[3] to automatically locate the introduced typed hole and to obtain a list of suggestions containing code elements that may fit. Let us consider Program 2 and its equivalent version with a typed hole represented by the underscore:

```
let f = fun x -> _ x (x + x)
```

Having introduced the typed hole, we can request a list of suggestions for the typed hole's location from the *OCaml LSP* and obtain 318 candidates. This list is not curated according to the type context and, as such, the suggested completions may present a type mismatch. In order to filter the list accordingly, type unification is performed between each element and the expression that was flagged as problematic according to the application of *typecast*. If unification succeeds, the suggestion will take part in the resulting list which, in this situation, will consist of 23 candidates.

3.2 Strategies

We make use of the available GPT-3 operation modes to implement three of the four repair strategies supported by our tool. Depending on which strategy we intend to use, we have to prepare and format the data accordingly.

3.2.1 Fill. GPT-3 provides an operation mode named *Insert*, in which, given some input text from the user which contains a hole denoted by the `[insert]` tag, a generation is produced by the model and placed where the tag was located. Thus, this operation mode is perfect for our use case, by filling programs in which a part is missing. There are several models available for this operation mode, notably `text-davinci-003` and `code-davinci-002`, the former being a general model and the latter being optimized for handling code. Next, we show an example of an input prompt for this operation mode:

```
let rec add_list lst = match lst with
  | [] -> [insert]
  | fst :: rest -> fst + (add_list rest)
```

For this prompt, using the web interface and with the default model parameters, both `text-davinci-003` and `code-davinci-002` models will output the following:

[3] https://github.com/ocaml/ocaml-lsp

```
let rec add_list lst = match lst with
  | [] -> 0
  | fst :: rest -> fst + (add_list rest)
```

For this program, we obtain the correct patch. Of course, to do so we need to first locate the error and replace it with the `[insert]` tag. We do this through the technique described in Subsection 3.1.1, by replacing the code that did not typecheck without the usage of the *typecast* function with said tag. This is the strategy in which we provide the least information to the GPT-3 model.

3.2.2 Choose. GPT-3 provides an operation mode named *Complete*, in which, after being fed input text from the user, it will attempt to generate more text based on it, that is, complete it. It can be used for non-code tasks such as writing stories or classifying tweets, as well as code tasks like translating plain text to an SQL query. We experimented with several approaches for usage of the *Complete* mode, because, unlike with the other operation modes, there is no intuitive way to use this mode to correct programs. Failed approaches include asking the model to rewrite the entire program replacing the missing hole (similar to the `[insert]` tag mentioned in Subsection 3.2.1) with the correct solution, or asking the model to just give us the code expected in that hole. Variations of this approach, by providing more clues, such as the expected type of the result, or by providing possible solutions to consider, were also unsuccessful. Ultimately, we were able to obtain favourable results by formatting the input as an exercise, similar to what would be found in a student exam. To do this, we present the source code with a missing hole denoted by the `<mask>` identifier, and a list of possible solutions. This list is produced as described in Subsection 3.1.3 and presented as numbered options, and the model is asked to select the most appropriate. We guide the model into selecting one option through *prompt engineering*. Specifically, every produced prompt is preceded with two example exercises that share this template but have the correct option selected (omitted in listings for brevity). The following program is an example of a prompt formatted for the *Complete* operation mode, as described.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> <mask>
 | fst :: rest -> fst + (add_list rest)

Which of the following options should replace <mask>?
1) ( __LINE__ )
2) ( max_int )
3) ( min_int )

Correct option:
```

Notice that all presented options are incorrect - this is a limitation from using this kind of prompt. Because we are using the *OCaml LSP* to generate suggestions to be then presented here, we are limited in which suggestions can be included. In fact, the *OCaml LSP* will not generate common constant values such as `0`, which is the correct response here. All the listed suggestions are integer constants suggested by the *OCaml LSP*, where `__LINE__` is a compiler macro representing the code line number where it is written, and `max_int` and `min_int` are constants representing the maximum and minimum values possible to represent as integers in *OCaml*. The following is another prompt (re-formatted for brevity) we produce for the same program, but assuming an error in a different place.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> []
 | fst::rest -> <mask> fst (add_list rest)

Which of the following options should replace <mask>?
1) ( fst )              8)  ( raise_notrace )
2) ( ! )                9)  ( snd )
3) ( exit )             10) ( @@ )
4) ( failwith )         11) ( max )
5) ( input_value )      12) ( min )
6) ( invalid_arg )      13) ( List.cons )
7) ( raise )            14) ( @ )

Correct option:
```

Notice that the list of suggestions grew — some of them, such as `exit` and `raise`, will match a lot of types, due to the polymorphic nature of these suggestions. However, some interesting suggestions are now listed, and in this case, the model suggests option **14** - the `@` operator. This operator concatenates two lists, and placing it into the hole in the source code will transform this function into a correct implementation of the `List.concat` function for joining a list of lists of values into a single list of values. It is, however, not what we tend to expect out of a function named *add_list*.

Communication with GPT-3 for this operation mode is fairly similar to other modes, but we have to limit the number of generated tokens, as the model tends to try to generate an explanation for its answer. It is also possible to specify a *stop sequence*, which is a sequence of tokens that, when generated by GPT-3, stops the whole generation process.

3.2.3 Instruct. Another way of interacting with GPT-3 is through its *Edit* mode which expects two inputs: a prompt and instructions describing how to edit the prompt. Similarly to the other modes, there is a more general textual model and a code specific variant. However, for this mode, there are specialized versions to handle text editing, namely `text-davinci-edit-001` and `code-davinci-edit-001`. Our approach uses a simplified form of the *Instruct* mode, which is applied when the step in Section 3.1.1 fails to produce a program that typechecks. In that case, the prompt consists of the original program, and the instruction will hold the message "*Fix the bug*". Alternatively, in case the previous step is able to produce a well-typed program[4], our approach

[4]Recall that whenever bypassing the type system by using the *typecast* function eliminates the type error, we explore that program variant.

performs inlining and type unification on the *typecasted program* in order to compute the minimal substitution holding the expected type with as much information as possible from the whole program. If we consider Program 1, the inputs sent to GPT-3 would be:

```
Prompt:
let rec add_list lst = match lst with
  | [] -> _
  | fst::rest -> fst + (add_list rest)
Instruction:
Replace the underscore with something of type int
```

The hole represented by the underscore is the place we wish to see filled in. Although the underscore character can appear in an *OCaml* program, we did not notice any interference in the ability of GPT-3 to apply the transformation in the intended place. The template we use for the edit instructions is "*Replace the underscore with something of type* `<inferred>`". For this program, GPT-3 responds with 0, which is the desired fix. Indeed, it may look like GPT-3 simply understands the program in question is missing the most adequate stop criteria and just answers with a corrected version, perhaps disregarding our instructions. However, the unification and inference step we perform in order to complete the message template with the expected type plays a crucial role. For the same example, fabricating messages referring illogical types such as *string* or *('a → 'b) → 'a list → 'b list* would see GPT-3 answer with the *empty string* and `List.map` respectively.

Because our approach explores every application of *typecast* that typechecks a program, we also produce another alternative:

```
Prompt:
let rec add_list lst = match lst with
 | [] -> []
 | fst::rest -> _ fst (add_list rest)
Instruction:
Replace the underscore with something of type 'a -> 'b
     list -> 'b list
```

Even though this alternative prompt will not generate the intended fixed program, it shows that the creation of adequate prompts is essential for GPT-3 to perform well. In this case, GPT-3 will respond with `(fun x y → x::y)`. Surely, integrating that piece of code into the original program produces a correct one from a typechecking perspective, although it does not fulfill the programmer's intention.

3.2.4 Without GPT-3. One interesting outcome from the implementation of the *Choose* strategy described in Section 3.2.2 is that we can make use of the work done to construct the prompt and skip the interaction with GPT-3. Thus, we provide a way to work completely offline. After coming up with an alternative program that typechecks and a list of suggestions (according to sections 3.1.1 and 3.1.3, respectively), we integrate each one into the original program. If no test cases have been provided, the tool simply displays which options fit the expected type. If there are test cases, the tool tests each suggestion and displays the resulting programs according to whether they satisfy the tests or not. Consider the following ill-typed program and the associated test case:

```
let f = fun x -> x && (x + x)
Test case: f 3 = 9
```

According to the test case, the intended fix consists of replacing the logical-and (&&) with the plus operator (+). For this case, our tool is able to filter 15 suggestions out of the 318 provided by the *OCaml LSP*, with one of them being the desired one. Each of the 15 suggestions is checked against the test case and the tool outputs the only program that satisfies the criteria:

```
let f = fun x -> (+) x (x + x)
```

Note that the type unification step requires the use of functions. In that sense, we convert the usage of operators such as '&&' to their equivalent prefix notation functions '`(&&)`', resulting in the generated patches also being written in this form, demonstrated by the use of the function '`(+)`'.

Indeed, the focus of our work is to evaluate GPT-3's performance regarding the automatic repair of type errors, and presenting a method in which the usage of the model is non-existing may seem counter-intuitive. However, we find this to be a validation of our approach, showing that the effort to assemble the prompt can guide the whole process towards the intended result as the correct patch may be found by further checking each plausible option.

3.3 Model Bias

We now experiment with providing more information, trying to guide the models into more relevant results. We do this by using the *bias* parameter which lets us guide the model's output by specifying the importance of certain tokens[5] through weights. A token represents a unit of text, like a character or a word. We use a tokenizer tool for this purpose.

We create a database of the most common tokens in the top 10 *OCaml* repositories on *GitHub* programmatically. To achieve this, we utilize GPT-3's tokenizer, which converts text into numerical sequences that the model processes. We analyze the tokens in source code files from these repositories and collect frequency data to construct a database of commonly used tokens in real-world programs.

We create a list of suggestions as per Section 3.1.3, convert them into token sequences, and assign positive weightings to these tokens. Then, we use the *bias* parameter in GPT-3 to guide the model toward these suggestions. The weightings are determined based on a database of token frequencies from real-world programs. We heuristically set minimum and maximum bias values at 1 and 3, respectively, to ensure effective guidance without extreme behavior. Figure 2 provides an overview of this process with sample values.

[5]Tokenizer available at https://beta.openai.com/tokenizer

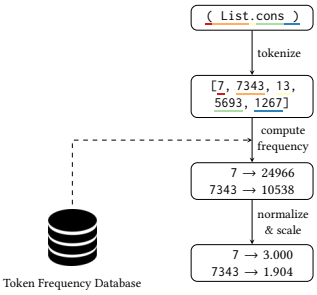

Figure 2. Bias computation for one *OCaml LSP* suggestion.

We experiment with *bias* values by comparing the *Choose* strategy with and without *bias*. In the *Choose* strategy with *bias*, we exclude suggestions from the textual prompt since their influence is already provided through *bias* values, as we show in the following example:

```
Consider the following OCaml program:

let rec add_list lst = match lst with
 | [] -> <mask>
 | fst :: rest -> fst + (add_list rest)

What should replace <mask>?

Answer:
```

3.4 Test Cases

Mentat allows including test cases when repairing a program. This additional information enhances the system's performance by narrowing the error search space and tightening the type constraints for the function under examination. To illustrate, recall Program 1, which contains two potential errors. Now, let's add a test case into this program.

```
let rec add_list lst = match lst with
  | [] -> []
  | fst :: rest -> fst + (add_list rest)

Test case: add_list [1;2;3] = 6
```

The newly added test case locks function `add_list` to specifically receive a list of integers and output a single integer. Because of this, the function now only has one possible source of error, which is the empty list (`[]`) in line 2. Previously, it was also considered that the *plus* operator (`+`) could be a source of error, but with the additional restrictions imposed by the test case, this is no longer possible.

Adding at least one test case to the framework also helps classify GPT-3's generations. After repairing a program, we can use the test case to check for type consistency and verify if it now passes the tests. For instance, in the case of this program, the correct fix would be to replace the empty list (‘[]‘) with the number 0, but substituting it with any other integer would still pass the type-check, although it might produce incorrect results during testing.

4 Tool

To validate our approach, we implemented it as a publicly available tool called Mentat [6]. This tool, written in *OCaml*, can analyze *OCaml* programs and is accessible via the command line. Users can specify:

- the file containing the *OCaml* program to analyze;
- the repair strategy by issuing the corresponding flag;
- optionally, one or more test cases that should be satisfied.

Depending on the repair strategy selected by the user, Mentat interacts with GPT-3 by calling the relevant function and setting appropriate parameters. Interaction with *OpenAI's* GPT-3 like models requires an internet connection to use the API. Mentat handles these requests and processes the responses to generate potential fixes for type errors. The resulting programs are saved for further offline analysis, including whether they compile successfully and pass provided test cases if available. Detailed installation and usage instructions are provided in the tool's repository.

5 Experiments

We benchmark the effectiveness of our tool by running it against several *OCaml* programs containing type errors. For this, we run each strategy 3 times for each program, and record the results. All the examples and necessary resources to replicate the experiments are publicly available[6].

5.1 Simple Programs

This set includes 15 ill-typed programs sourced from an introductory *OCaml* class at a Japanese University and the type-error slicer *Skalpel* [37], and previously used in a type-error debugger [50]. These programs are simple, with issues like returning empty lists instead of sums, confusion between `Float` and `Int`, and using values when singleton lists were expected. They range from 29 to 117 tokens and consist of 2 to 8 lines of code. One could argue that simple programs are easier to fix because they are simple, or harder to fix due to the limited contextual information available.

For the text and code models used in the experiments, we use the default parameters (*temperature* of 0.7 for text and 0 for code, and *top_p* value of 1 for both). These settings were found to be the most suitable through extensive testing.

We present the experiment results in Table 1. Each test program was processed 3 times to measure successful patch generation, ensuring it passed at least one test case. We employed different repair strategies with models optimized for *text* (T columns) and *code* (C columns). The *C + Bias* column includes additional experiments detailed in Section 3.3. The rightmost column represents results without language models, measuring how many suggestions enabled program compilation and passed a test case. For example, program *S2* was exclusively repaired by the code variant of the *Fill* strategy, with 10 successful repair suggestions that passed the test

[6] https://gitlab.com/FranciscoRibeiro/mentat

Table 1. Automatic repair results for 15 simple test programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
S1	3	3	3	3	3	3	3	3
S2	0	3	0	0	0	0	0	10
S3	3	3	0	0	3	3	3	3
S4	3	3	0	0	0	2	3	0
S5	3	3	0	0	3	3	3	2
S6	3	3	3	3	0	3	3	3
S7	3	3	0	3	3	0	0	0
S8	3	0	3	3	3	3	3	1
S9	3	3	3	3	0	0	0	1
S10	3	0	3	3	0	0	0	1
S11	3	3	0	0	3	3	3	0
S12	0	3	3	0	3	2	3	0
S13	3	3	3	3	3	0	0	1
S14	3	3	3	3	3	0	0	2
S15	3	3	3	3	3	3	3	1
%Repair	87%	87%	60%	60%	66%	56%	60%	–
%Test	53%	60%	47%	47%	40%	49%	53%	–

case. Further refinement may be possible by using different or additional test cases. In each column, we calculate two success rates: *%Repair*, indicating partial success (yellow or green), and *%Test*, indicating total success (treating yellow results as failures).

Each cell of the table is coloured red, yellow, or green. Red cells denote a total failure of patch generation, green cells denote the generation of the correct patch, and yellow cells denote partial success. Examples of patches that are categorized yellow include generating the incorrect arithmetic operator (such as generating a *minus* sign when a *plus* sign is expected, or not generating the correct constant value when one is expected) - in some cases, for example when a constant value is expected, it might be completely impossible to reasonably deduce which value the developer expects. Such results can be adjusted by using different test cases which favourably guide the GPT-3 model - for example, if, for a given test case, using a *plus* sign yields the same result as using a *minus* sign, perhaps changing the test case will make the usage of a different operator yield a different result. Nevertheless, we decided to not fine-tune the test cases to maximize result quality, as that is not always realistic.

The results showcased in Table 1 point towards the *Fill* strategy being the most efficient for automatic generation of patches. Most notably, all modes have a *%Repair* success rate above 50% and a *%Test* success rate above 40%, and all test programs were successfully repaired by at least one of the repair strategies. This fact points towards the combination of strategies being a robust approach to leverage the strengths of each other. We also denote that most cells contain the values 3 or 0, with rare occurrences of 2, which implies that the model tends towards the same results in different iterations. For this, we have experimented with different values of the parameters we supply to the model, focusing mainly on the *temperature* as it should change its randomness. Nevertheless, the results were not noticeably better, generally leading to lower overall success rate.

Table 2. Automatic repair results for 10 Dijkstra programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
D1	3	3	0	3	3	0	0	0
D2	1	3	0	0	0	0	0	0
D3	1	3	0	0	2	2	3	0
D4	3	3	3	3	0	1	3	1
D5	2	3	0	3	3	2	3	1
D6	3	3	0	0	0	0	0	0
D7	3	3	3	1	3	3	3	1
D8	3	3	0	0	3	3	3	0
D9	3	3	3	3	0	3	3	0
D10	3	3	0	0	3	0	0	0
%Repair	83%	100%	30%	43%	57%	47%	60%	–
%Test	83%	100%	20%	23%	20%	47%	60%	–

We observe that the usage of *bias* with the *Choose* operation mode yields relatively similar results in terms of success rates for these problems. The main difference when using *bias* lies in the fact that some programs that were not repaired with the previous approach are now able to be repaired and vice-versa. For this set of programs, we conclude that the usage of *bias* does not improve the results significantly, but it is capable of generating solutions complementary to the ones generated by the original *Choose* repair strategy.

5.2 Dijkstra Algorithm

In this set, we have longer and more complex programs for the *Dijkstra* algorithm, each with around 2,300 tokens and 170 lines of code. Deliberate errors were added to make the repairs more challenging. We followed the same methodology as in Section 5.1 for the results.

Table 2 summarizes the results for this program set. Like in the previous set (5.1), *Fill* remains the most effective strategy with an 83% repair rate for the *text* model and 100% for the *code* model. Despite the increased program complexity, *Fill* performed better, with a higher rate of programs passing the provided test cases. Conversely, the other strategies, *Choose*, *Instruct*, and *No GPT-3*, were less effective with this program set. Indeed, depending on the considered repair strategy, the discrepancies across the different sets of programs move in opposite ways. Increased program complexity may have improved *Fill*'s performance by providing more context for the model, while negatively affecting the other strategies, which seem more suited for shorter and simpler repairs.

Compared to the simpler programs in the previous section, the type errors in this set usually need more elaborate repairs. As an example, consider the ill-typed excerpt from a program contained in these experiments and its intended repair:

```
let rec search tree k = match tree with
  Empty -> raise Not_found
 | Node (left, key, value, right) ->
   if k = key then value
   else if k < key then left (* intended: search left k *)
   else search right k
```

Instead of `left`, the intended expression is `search left k`. These repairs need an aggregation of several terms, which is impossible to obtain with suggestions from the *language server*. Essentially, this severely hinders the *Choose* and *No GPT-3* modes, as they heavily rely on that list of code completions. The *Instruct* mode correctly infers the corresponding hole's type to be *(string * float) list*[7] but is unable to generate the call `search left k` and produces the empty list instead, which, nonetheless, produces a correctly typed program. From these experiments, we take that *Fill* works best for longer programs, as the pure context of the code seems to be enough and better allows GPT-3 to understand and reason about the program at hand. Extra analysis of the source code prior to providing the programs to GPT-3 is more helpful for smaller programs, in which naturally occurring context lacks. This is evidenced by programs $S4$, $S5$, $S8$, $S9$, $S10$ and $S13$, for which *Fill* presented incorrect or only partially correct results, while *Choose*, *Instruct* or *No GPT-3* were able to generate intended outcomes. This did not occur for the *Dijkstra* programs, as *Fill* showed that it could match the effectiveness of the other strategies for each case.

5.3 Large Scale Evaluation

We also conducted a large-scale evaluation of our approach. We analyzed a repository of 4,500 *OCaml* programs, which had already been created as part of Rite [41]. We provide detailed analysis of the results obtained from this evaluation, such as the total repair rate, the number of partially fixed programs and the distribution of effectiveness of the three repair strategies. Through this evaluation, we aim to demonstrate our tool's applicability in real-world scenarios and potential to improve the quality and reliability of large-scale software systems.

5.3.1 Pre-Processing the Data. To ensure a comprehensive and accurate evaluation of our tool, we applied a filtering process to the original dataset obtained from the Rite project. Specifically, we filtered out bugs that required modifications in multiple and disjointed places in the code, as the current version of our tool considers single expression bugs, only. Furthermore, we only considered bugs for which the original fixed version could properly execute for all test cases generated by the *OCaml* property-based testing tool *Quickcheck* [10]. Proper execution was defined as the absence of errors or timeouts for any given input. This was necessary to ensure that the bugs were genuine and that any improvements observed in our evaluation were a result of our tool's impact, rather than external factors such as faulty test cases or unreliable program behavior. After applying these filters, we evaluated a set of 1,318 bugs.

[7] Actually, the function is polymorphic, but the test case requires a more specialized type, which is what we get thanks to inlining.

5.3.2 Validating the Generated Patches. To validate the effectiveness of our tool in repairing bugs, we used *Quickcheck* to generate a random, large number of test cases. Moreover, we define properties to assert that the human-fixed program is "equivalent" to the repaired one. Thus, for each bug, we generated a set of patches and automatically instantiated a corresponding *Quickcheck* property. This is expressed according to the following template:

```
1 let%test_unit "testName" =
2   Quickcheck.test
3     [%quickcheck.generator: <input_signature>]
4    ~f:(fun args ->
5     [%test_eq: <output_signature>]
6     (Fix.functionToTest args) (Gen.functionToTest args))
```

To generate a property for the faulty program being repaired, we consider the faulty function's signature. The input part of the signature (line 3) is used to implement a generator for the input values that will be tested. The output part (line 5) is used to tell *Quickcheck* the type of the output values to compare. Line 6 represents the property that should be verified and means that the result of the original fixed program should be equal to the result of the patch being tested. The number of arguments needs to be adjusted according to the function being tested and, as such, `args` is modified to reflect that. By default, the generator produces 10,000 inputs. If the property holds, the patch is considered equivalent to the fixed version.

A bug is considered to be repaired if at least one of the generated patches produced the same output as the original fixed version for all input combinations generated by *Quickcheck*. By automating the instantiation of this property for every considered bug, we were able to accurately validate the effectiveness of our tool in repairing bugs. This approach also allowed us to provide quantitative metrics on the performance of our tool, such as the percentage of bugs repaired and the degree to which some bugs are partially fixed — Figures 3 and 4. This automated validation process is crucial given the amount of data at this stage. Furthermore, it also provides some insight into how it could be incorporated in real-world use cases. To the best of our knowledge, it is uncommon to provide a fully automatic validation process to verify whether generated patches successfully fix buggy programs. Our approach has the flexibility of allowing patches equivalent to the intended fix, without relying on human intervention to manually inspect the generated patches.

5.3.3 Results and Discussion. Our approach successfully repaired a substantial portion of the dataset. Among the 1,318 bugs evaluated, our tool repaired 516 of them, achieving a repair rate of 39.2%. We found that 441 of the programs were partially fixed, indicating that the generated patches were able to address some but not all of the identified issues in the program, representing a 33.5% partial repair rate. The consideration of partial fixes provides a more nuanced understanding of the capabilities of our technique. Rather than

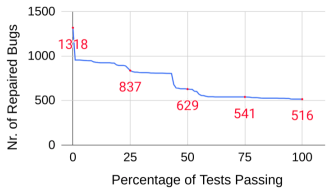

Figure 3. #Programs that pass at least a given percentage of tests. For example, 629 programs pass at least 50% of tests.

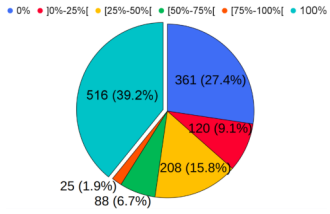

Figure 4. Distribution of test passing rate of programs. For example, 208 programs pass between 25% and 50% of tests.

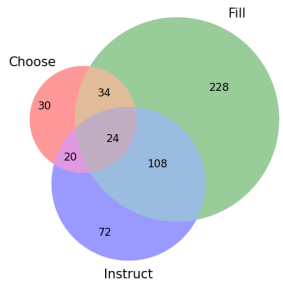

Figure 5. How many programs each mode successfully repairs. Intersections mean that a program is repaired correctly by both modes. There are 34 programs that can be repaired by both the Choose mode or the Fill mode.

simply categorizing a program as either fixed or not fixed, partial fixes enable us to explore the ground that separates a completely fixed program from a program that remains broken. Thus, we can form an idea of how the partial fixes are distributed along that spectrum. Out of the 361 programs that remained unfixed, we found that 247 of them produced some error during testing and the testing process did not finish. Additionally, 73 of the unfixed programs were due to our technique being unable to generate any patch for the identified bugs. Interestingly, we also found that 41 of the unfixed programs actually failed every test produced by *Quickcheck*, indicating that the bugs in these programs were particularly challenging to address.

Different tool modes exhibit varying degrees of repair effectiveness, as shown in Figure 5. The *Fill* strategy is the most effective, being responsible for fixing 394 out of the total 516 programs (76.4%). The *Instruct* strategy was also found to be effective, repairing 224 programs (43.4%). On the other hand, the *Choose* strategy is the least effective, with 108 fixed programs (20.9%). It is worth noting that some programs were repaired by multiple strategies, and in some cases, the same program was repaired by all three strategies. Specifically, there were 108 (20.9%) programs that were fixed by both *Fill* and *Instruct*, while 34 (6.6%) programs were fixed by both *Fill* and *Choose*, and 20 (3.9%) programs were fixed by both *Choose* and *Instruct*. Additionally, there were 24 (4.7%) programs that were repaired by all three strategies.

5.3.4 Limitations.

Our automated validation strategy excludes functions relying on user-defined data types, as it needs manually defined specific generators. This limitation reduces the number of programs we analyze, as discussed in Section 5.3.1. Moreover, we assume total functions, meaning that we consider every possible input for a given type, resulting in a more pessimistic repair validation. For instance, if a function has an integer as argument and is designed to work only with positive numbers, our fully automated approach will still test it with negative numbers (as produced by the predefined generator of integer numbers) reporting it as a non repaired function.[8]

Let us consider the *OCaml* implementation for *factorial*:

```
(* int -> int *)
let rec factorial n =
 if n = 0 then 1
 else n * factorial (n - 1)
```

The provided implementation is the usual recursive definition for *factorial*. Note this is a partial function as it is only defined for positive values of the input `n`. If `n` is a negative number, *factorial* will indefinitely call itself causing a stack overflow error. Now, let us consider that this implementation of *factorial* results from a repair process, either generated by Mentat or another tool. When we validate such repair with our automated validation approach, we use *Quickcheck* to automatically generate inputs for this function. In this case, the predefined generator for *int* will produce both positive and negative values. Although the repaired *factorial* function is correct, our validation will fail due to timeout as soon as it is called with a negative number.

5.4 Comparative Study

We performed a comparative study of our technique for automated program repair of ill-typed *OCaml* programs. We utilized the results provided in Rite's [41] repository for both their tool and Seminal to validate the efficacy of our fully automated validation strategy. In section 5.3.4, we acknowledge the demanding and pessimistic nature of our testing strategy by highlighting its consideration of total functions, encompassing every possible input for any given type. This ignores any restriction on the set of valid inputs. A manual validation process, similar to that employed by Rite, has the potential to increase the success rate for both

[8] Generators for positive integers and user-defined types can be implemented. However, this would break our goal of a fully automated process.

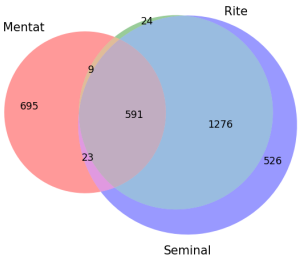

Figure 6. #Programs used in each repair technique and intersections.

our approach and the others. This kind of manual validation allows for a more sensitive consideration of program characteristics that may be overlooked by a more automated validation method. That is, some expected usage patterns may be better captured by a human evaluator with a more subjective evaluation criteria. An example is judging a function's implementation and considering it has been designed to only work with positive numbers, even though the type system may only reflect the function operates on type *int*. However, such manual validation would imply extensive manual effort and is infeasible for the size of this dataset.

We compare our approach with two other tools, namely Rite and Seminal, in terms of their repair capabilities on a common dataset. Although the three tools used a common dataset as an underlying basis, each work applied its own preprocessing criteria to prepare the dataset. As a consequence, in this comparative study, our original dataset of 1,318 bugs was filtered down to 591 bugs, which were common to all three approaches. Figure 6 shows the distribution of the bugs and how they intersect among Mentat, Rite and Seminal.

Our technique achieved a repair rate of 37.6% (222 out of 591 programs). It employs a fully automated analysis that considers a program fixed only if it becomes well-typed and passes all test cases. Our repair process leverages GPT-3, a powerful large language model, to generate patches for identified type errors. This eliminates the need for a comprehensive system and language-specific components due to GPT-3's extensive training on multiple languages.

Originally, Rite conducted a manual validation through a user study with 29 programmers in which a set of 21 buggy programs was selected and each participant was shown 10 randomly selected buggy programs alongside two candidate repairs, one generated by Rite and one by Seminal. A full validation of the entire dataset was not reported. To achieve this, we used our automated validation framework to verify which Rite and Seminal generated patches were able to pass all test cases produced by *Quickcheck*.

This way, we were able to evaluate the performance of Rite and Seminal on the same dataset. Rite repaired 198 programs out of 591 (33.5% repair rate), while Seminal repaired only 46 programs (7.8% repair rate). These results highlight the superior effectiveness of our technique over the existing

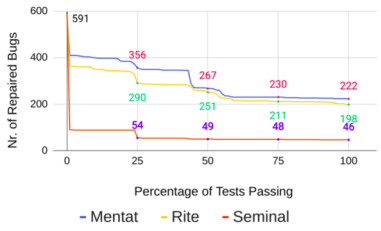

Figure 7. Number of programs that pass at least a given percentage of tests - comparative study.

state-of-the-art tools for automated program repair in the context of type errors in *OCaml* programs. Figure 7 shows the repair effectiveness of the three tools.

One noteworthy advantage of our approach is its language-agnostic nature. Our technique can be easily adapted to repair programs in other languages, as long as it is possible to statically determine the types of terms either through inference or annotations, and the ability to bypass the type system exists (e.g., `Obj.magic` for *OCaml* or `undefined` for *Haskell*). Furthermore, the reliance on LLM's, such as GPT-3, for generating patches liberates us from building language-specific generation systems for each case. By leveraging these prerequisites, our approach can be successfully applied to a wide range of programming languages.

We conclude that Mentat outperforms both Rite and Seminal in repairing type errors on a common dataset of *OCaml* programs. Our fully automated approach eliminates the need for user studies to validate patch relevance and ensures that the resulting programs are not only well-typed but also pass all the provided test cases.

Our results provide the following four insights: First, Mentat surpasses both Rite and Seminal in terms of effective program repair, i.e. patches are well-typed and are equivalent to the intended fixed version; Second, we thoroughly validated Rite's repairs, whereas their paper only validates 21 repairs with user involvement; Third, although Rite reports over 80% success in type repair, we show that the percentage of repairs passing the tests is 33.5%, which is significantly lower and highlights the potential for misleading results[9]; Fourth, our fully automated validation approach enabled us to validate other works that previously relied on manual analysis of a very limited subset of programs.

6 Related Work

Type error debugging research has a rich history spanning over 30 years, evolving from enhancing error messages [12, 24, 51] to interactive debugging tools [6, 7, 9, 48], and automated approaches that narrow down error causes [19, 37, 42–44]. These methods aim to pinpoint errors and require user intervention for correction. On the other hand, automatic correction of type errors is a nascent field; Seminal [25] is,

[9]This also contradicts the (informal) usual saying in functional programming: if it type checks, then it is correct.

to our knowledge, the first system for automatic correction of type errors in functional programming languages. It removes parts of the ill-typed program and attempts to make syntactic changes. This corresponds to *Fill* in our study: they used a syntactic modification to fill, and we used GPT-3. Rite [41] aims for program repair of ill-typed programs too. From a corpus of 4,500 ill-typed *OCaml* programs, it uses approximately half of the dataset to build a neural network that learns what modifications have been made to, ultimately, synthesize solutions for given ill-typed programs. Our tool, Mentat performs source code analysis to produce useful prompts that leverage GPT-3's language understanding and generation capabilities to generate potential patches. While Mentat, Rite and Seminal share a common objective of fixing ill-typed *OCaml* programs, they diverge in their validation methodologies. Rite relies on a manual analysis of 21 randomly selected programs from the repository by a limited number of programmers, whereas our technique employs a fully automated process to validate the generated repairs. This distinction allows our technique to perform validation on a larger scale, effectively addressing the challenges associated with manual validation processes. The definition of a fixed program in Rite is based on the ability of the generated program to typecheck correctly. In contrast, our work validates both typechecking and semantical equivalence of the generated repairs. To achieve this, our technique employs a methodology that generates and executes test cases for both the correct program and the generated repairs. It considers a program to be fully repaired only if the correct program and the repaired version produce identical outputs for all test cases. This crucial difference allowed us to verify that a pure type repair can fall short of being an effective repair. We demonstrated that Rite's reported +80% type repair rate is comparatively lower in terms of actual program repair, i.e. the generated repair satisfies the test cases 33.5% of times. As we mentioned in Section 5.3.4, this is based on a pessimistic view that a patch must pass all test cases. Indeed, a manual analysis may reveal that more of the generated patches are semantically equivalent to the intended program, potentially improving our results as well as those of Rite and Seminal.

DeepTyper [22] enhances type information for compilation using deep learning in *Python* and *JavaScript*. However, it lacks program repair capabilities. Our work utilizes *OCaml*'s type inference for source code analysis and prompt preparation. DeepTyper could be beneficial when extending our approach to other programming languages.

Fault localization [2, 32] is an initial debugging step [31]. Various methods, including execution trace analysis [5], mutation testing [29], qualitative reasoning [33], and semantic fault identification [38], help narrow down suspicious code elements. Models like *code2vec* [1] have been trained to specifically detect security vulnerabilities [11]. Our work concentrates on type errors and uses *OCaml*'s type inference to identify potentially responsible expressions by transforming them into different types.

APR is a prominent research field. Early approaches use genetic programming [3, 23], while others employ constraint-based methods [16, 30, 52]. Recent advancements incorporate machine learning and neural machine translation techniques [8, 26, 28]. However, translating buggy code to fixed code has limitations [15] and general-purpose models supporting code understanding and generation tasks [1, 17, 27, 45] started being considered. GPT-2's code completion effectively fixes *Java* bugs [39], and *Codex* has repaired *Python* and *Java* programs [35]. Our work stands out for targeting type errors in *OCaml*, which prevent program compilation, unlike other research focused on functional bugs.

7 Conclusion

This paper introduced a method to automatically fix type errors in *OCaml* programs using GPT-3. We achieve this by analyzing and modifying the faulty source code to create prompts for GPT-3-based models.

We developed the Mentat tool, initially validating it with simple programs and variations of the *Dijkstra* algorithm. In large-scale experiments involving 1,318 buggy programs, we achieved a 39% repair rate using a novel automated patch validation approach. In comparison with two other *OCaml* program repair tools, Mentat outperformed them, achieving a 37.6% repair rate on a shared dataset of 591 programs, while the other tools achieved rates of 33.5% and 7.8%, respectively.

This work used GPT-3, but future versions or other LLMs [46, 53] could be integrated. Moreover, fine-tuning a model for *OCaml* may enhance program correction.

Starting with single-location faulty programs enables us to assess the approach's effectiveness before addressing multiple locations. In future work, we plan to explore this possibility by strategically placing typecast operators to address program sections and incorporating multiple typecasts in suitable locations. This would facilitate the identification and repair of multiple bugs within a single program.

Replication Package

All the necessary resources to replicate this study are public:

- **Tool:** *https://gitlab.com/FranciscoRibeiro/mentat*
- **Artifact [40]:** *10.6084/m9.figshare.23646903.v2*

Acknowledgments

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDP/50014/2020. Francisco Ribeiro and José Nuno Macedo acknowledge FCT PhD grants SFRH/BD/144938/2019 and 2021.08184.BD, respectively. Additional funding: JSPS KAKENHI-JP19K20248.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Aaron Ang, Alexandre Perez, Arie Van Deursen, and Rui Abreu. 2017. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 175–182.

[3] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Applied soft computing* 11, 4 (2011), 3494–3514.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[5] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*. 378–381.

[6] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *Symposium on Principles of Programming Languages. Proceedings (POPL '14)*. ACM, 583–594. https://doi.org/10.1145/2535838.2535863

[7] Sheng Chen and Martin Erwig. 2014. Guided type debugging. In *Functional and Logic Programming. Proceedings (LNCS 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

[8] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[9] Olaf Chitil. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *International Conference on Functional Programming. Proceedings (ICFP '01)*. ACM, 193–204. https://doi.org/10.1145/507635.507659

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

[11] David Coimbra, Sofia Reis, Rui Abreu, Corina Păsăreanu, and Hakan Erdogmus. 2021. On using distributed representations of source code for the detection of C security vulnerabilities. *arXiv preprint arXiv:2106.01367* (2021).

[12] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages. Proceedings (POPL '82)*. ACM, 207–212. https://doi.org/10.1145/582153.582176

[13] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as translation: the data and the metaphor. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 275–286.

[16] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

[19] Christian Haack and Joe B. Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming* 50, 1-3 (2004), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

[20] BJ Heeren, JT Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer. 2002. Improving type-error messages in functional languages. (2002).

[21] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 62–71.

[22] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[24] Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (1998), 707–723. https://doi.org/10.1145/291891.291892

[25] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. *SIGPLAN Not.* 42, 6 (jun 2007), 425–434. https://doi.org/10.1145/1273442.1250783

[26] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[28] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[29] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[31] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[32] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[33] Alexandre Perez, Rui Abreu, and IT HASLab. 2018. Leveraging Qualitative Reasoning to Improve SFL.. In *IJCAI*. 1935–1941.

[34] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[35] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: a constraint-based type error slicer for Standard ML. *Journal of Symbolic Computation* 80, 1 (2017), 164–208. https://doi.org/10.1016/j.jsc.2016.07.013

[38] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2021. On Understanding Contextual Changes of Failures. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 1036–1047.

[39] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2022. Framing Program Repair as Code Completion. In *Proceedings of the Third International Workshop on Automated Program Repair* (Pittsburgh, Pennsylvania) *(APR '22)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3524459.3527347

[40] Francisco Ribeiro, José Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair (SLE 2023). (10 2023). https://doi.org/10.6084/m9.figshare.23646903.v2

[41] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005

[42] Thomas Schilling. 2012. Constraint-free type error slicing. In *Trends in Functional Programming. Proceedings (LNCS 7193)*, Ricardo Peña and Rex Page (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[43] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Workshop on Haskell. Proceedings (Haskell '03)*. ACM, 72–83. https://doi.org/10.1145/871895.871903

[44] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. In *Workshop on Haskell. Proceedings (Haskell '04)*. ACM, 80–91. https://doi.org/10.1145/1017472.1017486

[45] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[47] Kanae Tsushima and Kenichi Asai. 2012. An embedded type debugger. In *Symposium on Implementation and Application of Functional Languages*. Springer, 190–206.

[48] Kanae Tsushima and Kenichi Asai. 2013. An embedded type debugger. In *Implementation and Application of Functional Languages. Proceedings (LNCS 8241)*, Ralf Hinze (Ed.). Springer, 190–206. https://doi.org/10.1007/978-3-642-41582-1_12

[49] Kanae Tsushima and Kenichi Asai. 2014. A weighted type-error slicer. *Journal of Computer Software* 31, 4 (2014), 131–148.

[50] Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. 2019. Type debugging with counter-factual type error messages using an existing type checker. In *Symposium on Implementation and Application of Functional Languages. Proceedings (IFL '19)*. ACM, Article 7, 12 pages. https://doi.org/10.1145/3412932.3412939

[51] Mitchell Wand. 1986. Finding the source of type errors. In *Symposium on Principles of Programming Languages. Proceedings (POPL '86)*. ACM, 38–43. https://doi.org/10.1145/512644.512648

[52] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

Received 2023-07-07; accepted 2023-09-01