



# Large Language Models in Automated Repair of Haskell Type Errors

Sofia Santos  
HASLab/INESC TEC  
University of Minho  
Braga, Portugal  
sofia.r.santos@inesctec.pt

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

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

## ABSTRACT

This paper introduces a new method of Automated Program Repair that relies on a combination of the GPT-4 Large Language Model and automatic type checking of Haskell programs. This method identifies the source of a type error and asks GPT-4 to fix that specific portion of the program. Then, QuickCheck is used to automatically generate a large set of test cases to validate whether the generated repair behaves as the correct solution. Our publicly available experiments revealed a success rate of 88.5% in normal conditions. However, more detailed testing should be performed to more accurately evaluate this form of APR.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software post-development issues*.

## KEYWORDS

automated program repair, large language model, fault localization, code generation, type checking, automatic testing

### ACM Reference Format:

Sofia Santos, João Saraiva, and Francisco Ribeiro. 2024. Large Language Models in Automated Repair of Haskell Type Errors. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24)*, April 20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3643788.3648012>

## 1 MOTIVATION

Automated program repair (APR) is an emerging technique that tries to automatically identify and suggest fixes for a bug in a given program. The goal of APR is to aid developers in the process of debugging, reducing their burden by automatically generating a patch for a buggy program [4].

This process can be divided into three basic processes: **Fault localization**, which attempts to find the code that must be modified to fix the bug, **patch generation**, which alters the code in order to produce a fixed version, and **patch validation**, which validates the previously generated code to ensure that the patch is a valid fix

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

APR '24, April 20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0577-9/24/04

<https://doi.org/10.1145/3643788.3648012>

for the program [5]. In this context, a patch is an altered or added piece of code that fixes the faulty program.

With the recent popularity of OpenAI's ChatGPT, Large Language Models (LLMs) have entered the public spotlight, but they were already being used for code generation and completion, with models created specifically for this task [11].

MENTAT [8] successfully used LLMs to generate patches for ill-typed *OCaml* programs. The strategy leveraged *OCaml*'s type system to identify candidate locations for the type error in the source code. Each candidate was replaced by an *insert tag* which enables GPT-3's *infilling* capabilities to generate alternative expressions in order to fix the program.

For this paper, we employed a similar approach for Haskell programs. Since Haskell and *OCaml* are statically typed languages, compilers for these languages can already detect type errors during the compilation stage. In fact, Haskell's type system can be seen as a fault localizer.

To better illustrate this advantage of statically typed languages, let's consider the following Haskell example:

```
addOne :: Int -> Int
addOne x = x + '1'
```

This simple function contains a type error. More specifically, it's trying to add an integer and a character. Since we know that the return type of the function must be `Int`, the culprit must be the character `'1'`. This can be verified by trying to compile the program.

```
* Couldn't match expected type `Int' with actual type `Char'
* In the second argument of `(+)', namely `1'
  In the expression: x + '1'
  In an equation for `addOne': addOne x = x + '1'
|
2 | addOne x = x + '1'
  |                      ^^^
```

The compiler was able to locate and to identify the source of the type error. By knowing where the error is, it's then possible to ask an LLM to generate a patch for that specific location.

Haskell's compiler, GHC, already contains a feature to fix possible type errors, known as type holes [3]. In the above example, by replacing the character with a type hole (represented by an underscore) and then compiling the program, GHC tries to guess possible values to fill the hole.

```
* Found hole: _ :: Int
* In the second argument of `(+)', namely `_ '
  In the expression: x + _
  In an equation for `addOne': addOne x = x + _
* Relevant bindings include
```

```

    x :: Int (bound at test_files\ex5\original.hs
:2:8)
    addOne :: Int -> Int (bound at test_files\ex5\
original.hs:2:1)
Valid hole fits include
    x :: Int
    maxBound :: forall a. Bounded a => a
    minBound :: forall a. Bounded a => a
|
2 | addOne x = x + _
|

```

However, this approach is limited, due to the fact that GHC only suggests functions or known variables. In other words, it doesn't suggest compound expressions (e.g.: `f (g x)`) or constant values, for example. In this case, the correct hole fit would be the number 1. The only possible way to know this would be by reading the function name and deduce that the value to add must be 1. As opposed to more traditional methods of APR, an LLM can infer a function's objective based on factors like its name. GPT-4, for example, due to its very large training data, can fix a function even if it has never seen that specific function before. LLMs can also generate any kind of expression, unlike GHC with type holes.

## 2 PROBLEM

The main question that this paper tries to answer is: *Could LLMs be used effectively in the process of automated program repair of Haskell type errors?*

Although there are already some solutions for performing APR of Haskell programs [2, 3], more specifically to generate possible fixes, the use of a language model could allow for a deeper understanding of the program, and therefore a more efficient method of patch generation for APR.

## 3 APPROACH

Since the goal in this paper is to verify the efficacy of LLMs in the Patch Generation process of APR, the two remaining steps, Fault Localization and Patch Evaluation, would have to be manually executed.

### 3.1 Fault Localization

Haskell's compiler GHC can identify type errors. However, it can't always locate the exact expression that is causing the type error. To give a specific example, this function contains a type error:

```

mysum :: Num a => [a] -> a
mysum [] = 0
mysum (h:t) = h ++ mysum t

```

Here, the list concatenation (`++`) function should be replaced with the arithmetic addition (`+`) function, since the objective is to add numbers, not join lists. When trying to compile this program, GHC points to the entire expression `h ++ mysum t` as the fault of the type error, which is not wrong, but could be more specific.

```

* Couldn't match expected type `a' with actual type
`[a0]'
* In the expression: h ++ mysum t
   In an equation for `mysum': mysum (h : t) = h ++
mysum t
* Relevant bindings include
  t :: [a] (bound at test.hs:3:10)
  h :: a (bound at test.hs:3:8)

```

```

mysum :: [a] -> a (bound at test.hs:2:1)
|
3 | mysum (h:t) = h ++ mysum t
|
   ^^^^^^^^^^^^^

```

Therefore, in order to locate type errors more precisely, we developed an algorithm that, for a given Haskell program, generates  $N$  variants. In each variant, one of the expressions (i.e., a function name, a variable name, a constant value, etc.) is replaced with the value undefined.

In Haskell, the value undefined is special because its type is a <sup>1</sup>. Therefore, it can be placed anywhere within the code and the program will still compile, it just might not execute accordingly. This property can be applied here to check for the specific location of a type error in a program.

By generating  $N$  variants, each with a different expression replaced with undefined, at least one of these variants won't contain the expression that was causing the type error, and that variant will therefore successfully compile.

To generate these variants, we developed a program in Haskell that recursively navigates through the abstract syntax tree (AST) representation of the faulty program and replaces any expression that it finds with undefined. Then, each AST is converted back into Haskell code and this code is written to a file.

Then, every generated variant is compiled. If a variant successfully compiles, we know that, in that variant, the type error has been removed. The location of the undefined expression within that variant's code is the probable location of the type error in the original program.

Sometimes, more than one variant can compile successfully. For the example shown in Listing 1, two variants can be produced:

```

tails :: [a] -> [[a]]
tails [] = [[]]
tails l = l ++ tails (tail l)

```

Listing 1: Original program

```

tails :: [a] -> [[a]]
tails [] = [[]]
tails l = undefined ++ tails (tail l)

```

Listing 2: Variant 1

```

tails :: [a] -> [[a]]
tails [] = [[]]
tails l = l `undefined` tails (tail l)

```

Listing 3: Variant 2

By either replacing the undefined in the the variant from Listing 2 with `[]` or in the variant from Listing 3 with `:`, we get a fixed version of the program. Despite this, there are some cases where multiple valid variants can be generated but only one of those will lead to a valid fix. During the Patch Evaluation step, we'll be able to identify such cases and keep only the valid fixed programs. Although these false positives can occur, that's not the case for false negatives. In other words, if a type error exists, it can always be identified with this strategy.

<sup>1</sup>In Haskell, polymorphic types are referred to with a lowercase variable name, most commonly a letter like `a` or `b`.

### 3.2 Patch Generation

To perform patch generation we rely on the GPT-4 LLM. GPT-4 is used to generate the most likely patch that will fix the program. It is given each of the valid variants generated in the previous step, with the undefined statement being replaced with a special `<INSERT>` tag. This replacement is done to make the prompt more natural, which helps to produce better results [6]. Listing 4 shows the prompt used to generate a patch using GPT-4. The system message contains the request and the user message contains the code.

```
{
  "role" => "system",
  "content" => "Fill in the <INSERT> tag. Answer must
    only contain the inserted text."
},
{
  "role" => "user",
  "content" => "f :: Int -> Int\nf x = x + <INSERT>"
},
```

**Listing 4: Example of a request sent to GPT-4 for the patch method.**

The two main objectives here are to give GPT-4 as little context as possible and check if it's able to produce a valid patch, while asking it to only reply with the patch itself, so it can respond faster and with fewer tokens.

After GPT-4 replies, the patch is then inserted into the respective variant, which still needs to be evaluated before the program can be considered "fixed".

### 3.3 Patch Validation

In order to automatically validate the generated patches, we use QuickCheck, a property-based testing framework commonly used to automatically test Haskell programs [1]. Quickcheck offers two key ingredients that are important for our patch validation: generation of a large set of random inputs (test cases), so that we are able to execute the produced variants, and the definition of properties, so that we can define a simple property that verifies whether the patch produces the same results as the (available) correct version of the function.

For this process, we need two files: a "buggy" program and a validator. The validator is a Haskell file that contains the correct version of the program and a QuickCheck property function that compares the output of a patched program with the correct version. If they are the same, then the patched version is labeled as a fixed version of the program.

## 4 EXPERIMENTS

After the main program was built, some experiments were conducted with a small sample of Haskell programs. All of these programs are simple, with only one or two functions, due to the difficulty of finding a large repository of complex Haskell programs that contain only type errors. A QuickCheck validator file was manually written for each of these tests<sup>2</sup>.

<sup>2</sup>Automatic generation of this file was considered, but due to the small amount of tests, it was faster to create them manually. However, it could be useful for future testing with a larger sample of tests.

Since GPT-4 and other LLMs have proven to be very efficient tools at fixing programs by themselves [10, 11], we also tested GPT-4's ability to fix type errors without telling it where the error was. In other words, for each test, we also sent GPT-4 the original version of the buggy program and asked it to fix the code, without the `<INSERT>` tag. These tests can show us if the additional effort of identifying the potential type error location provides a significant advantage, in comparison with just asking GPT-4 to fix the error.

In total, 26 tests were conducted. The code needed to run these tests, as well as the test programs themselves, are publicly available<sup>3</sup>. As mentioned above, there was some difficulty in finding adequate programs for testing. Therefore, these results should not be used as definitive evidence for any kind of conclusion, only as a starting point for more detailed validation.

## 5 RESULTS

For the method where we ask GPT-4 to generate a patch that replaces the special tag in the program, which will be referred to in this section as the patch method, 23 of the tests were successful. In other words, this method was able to fix 88.5% of the faulty programs by generating at least one patch that passed the evaluations. For some tests, more than one correct patch was able to be generated. However, to save resources and time, each test stopped once the first correct patch was found. Further testing could also include relevant statistics about these cases.

Curiously, the method where we just asked GPT-4 to fix the program without checking for the error location, which we'll call the fix method, also had an 88.5% success rate, which could indicate that both methods are equivalent. However, the cases where both approaches failed differed.

The programs that the fix method was unable to fix all had one thing in common: the function with the faulty code had the name "func". The patch method, on the other hand, was able to fix all of these programs, it only failed in other, more specific, cases.

```
type MSet a = [(a,Int)]

convertMSet :: MSet a -> [a]
convertMSet [] = []
convertMSet ((h,t):y) = t * h ++ convertMSet y
```

**Listing 5: Example of a function that the patch method was unable to fix.**

For the function above, the patch method was unable to fix the program, possibly because the type error lies in the `*` operator, which would need to be replaced by an infix version of the replicate function, which isn't very common in Haskell. Since the fix method has more liberty to rewrite the entire program, not just a small patch, it's able to rewrite the expression `t * h` as `replicate t h`.

These results could indicate that GPT-4 and other LLMs, when asked just to fix a program, are more efficient when they can determine what a function should do by its name. To validate this idea, we conducted a second round of tests, now with every function name being replaced with a generic name (e.g.: `f`, `g`, etc.).

<sup>3</sup><https://github.com/RisingFisan/Large-Language-Models-in-Automated-Repair-of-Haskell-Type-Errors>

**Table 1: Success rates of different APR methods.**

| APR Method                      | Success Rate (%) |
|---------------------------------|------------------|
| Patch                           | 88.5             |
| Fix                             | 88.5             |
| Patch (abstract function names) | 80.8             |
| Fix (abstract function names)   | 88.5             |

After this second round of testing, the results for the fix method were identical. However, the results for the patch method were worse than on the first set of tests. Now, only 21 programs were successfully fixed, a success rate of 80.8%. All five of these programs were successfully fixed when using the fix method.

From these results, it is safe to assume that the name of a function has an impact on how an LLM interprets the code, but it's more pronounced when the LLM can't see the faulty expression.

Despite its worse results, the patch method for APR might still be preferable in situations where the number of tokens generated is important, since the fix method needs to output at least the entire function that contained the error, as opposed to the patch method that only needs to output the patch. Furthermore, the fix method takes a longer amount of time to produce a result, since it needs to generate an entire program.

In these tests, the patch method generated an average of 2.15 tokens, with 14 out of 26 tests (54%) generating only a single token. Meanwhile, the fix method generated, on average, 51.23 tokens. This value is highly dependent on the size and complexity of the code for each test, but since these tests consisted only of one or two simple functions, the amount of tokens generated in real scenarios would likely be higher.

Another aspect to consider is that, since the fix method rewrites the entire program, it can change the formatting of the code, which might not always be desirable, or even the code itself in parts that did not need to be changed.

Ideally, a combination of these two methods might be ideal, since when one method failed a particular test, the other method passed, showing that they might be better suited for different kinds of programs.

## 6 FUTURE WORK

In the future, we aim to perform more tests with more complex programs. We also intend to experiment with different kinds of prompts. The current patch method just asks GPT-4 to insert the correct patch where the <INSERT> tag is. However, the results from the fix method indicate that GPT-4 might perform better when given the full context of the buggy program. Therefore, improving the prompt, by, for example, including the faulty code that was replaced by the tag might lead to better results of the patch method.

The current patch generation method only asks GPT-4 to produce one patch, but generating multiple patches and testing all of them could lead to improved results.

Despite this, these results already show us that GPT-4 is a very powerful tool. By discovering the best ways to interact with it, we can develop very efficient methods of APR.

It could also prove interesting to test this method of APR on other programming languages, more specifically dynamically typed programming languages. However, when dealing with languages like Python the process of finding type errors is more complicated [7], since only after executing the patched program can one know if the type error has been fixed. Hence, a different kind of fault localization method would be needed.

## Replication Package

All the necessary resources to replicate this study are public:

- **Repository [9]:** [10.6084/m9.figshare.24769038.v2](https://doi.org/10.6084/m9.figshare.24769038.v2)

## REFERENCES

- [1] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. <https://doi.org/10.1145/351240.351266>
- [2] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2016. Ask-elle: An adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2016), 65–100. <https://doi.org/10.1007/s40593-015-0080-x>
- [3] Matthias Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. 2022. PropR: Property-Based Automatic Program Repair. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1768–1780. <https://doi.org/10.1145/3510003.3510620>
- [4] 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>
- [5] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. <https://doi.org/10.1016/j.jss.2020.110817>
- [6] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2023. GPT understands, too. *AI Open* (2023). <https://doi.org/10.1016/j.aiopen.2023.08.012>
- [7] Wonseok Oh and Hakjoo Oh. 2022. PyTER: Effective Program Repair for Python Type Errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 922–934. <https://doi.org/10.1145/3540250.3549130>
- [8] 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 (Cascais, Portugal) (SLE 2023)*. Association for Computing Machinery, New York, NY, USA, 111–124. <https://doi.org/10.1145/3623476.3623522>
- [9] Sofia Santos. 2023. Large Language Models in Automated Repair of Haskell Type Errors. (12 2023). <https://doi.org/10.6084/m9.figshare.24769038.v2>
- [10] Nigar M. Shafiq Surameery and Mohammed Y. Shakor. 2023. Use Chat GPT to Solve Programming Bugs. *International Journal of Information Technology & Computer Engineering (IJITC)* ISSN : 2455-5290 3, 01 (Jan. 2023), 17–22. <https://doi.org/10.55529/ijitc.31.17.22>
- [11] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (San Diego, CA, USA) (MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/3520312.3534862>