Improving Error Messages in Ocaml for Novice Programmers

SCARLETT XU

When novice programmers are exposed to a strongly typed language like OCaml, learning how type checker works and the meaning of cryptic type error messages is a major obstacle to debug type errors successfully. Imprecise error locations and confusing explanation are two main problems of OCaml error messages, which can shift our attention to somewhere else instead of the actual source of the type error. Therefore, in order to eliminate the struggle with debugging type errors, we decide to explore some feasible modifications on the compiler to improve type error messages such that beginners can have a better picture of both the reasons behind type conflicts and corresponding fixing suggestions.

By providing an entry level tool for debugging type errors in OCaml, we aim to promote functional programming education, smooth beginners' learning curve, and make learning typed functional programming more accessible to novice programmers alike.

Additional Key Words and Phrases: debugging, error messages, type checking, functional programming

ACM Reference Format:
Scarlett Xu. 2020. Improving Error Messages in Ocaml for Novice Programmers. 1, 1 (August 2020), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A type error is often regarded as the conflicts between the types of two expressions. Specifically for a strongly typed language like OCaml, the compiler would conduct type checking before transform the AST (Abstract Syntax Tree) into intermediate representation. The existence of type checker ensures that all operations performed on a value are allowed by the type of that value. If the source code fails to pass the type checker during compilation, it would provide one piece of type error message at a time which explains the type conflict reason and location. Correct and informative type error messages can help programmers fix the error. However, it is quite challenging for novice programmers, who have little knowledge of the type inferencing algorithm, to understand the meaning of them before coming up with any fixing solution. The problems of current system mainly manifest in two aspects: (1) Error locations can be imprecise. (2) The explanation of type conflicts is irrelevant. By showing specific examples concerned with intrinsic properties of OCaml, such as strict constraints for expressions, the weaknesses of current type error messages gradually surface. In order to address them and encode easy-to-read message content, we choose an expression-based method to implement all modifications on the typing of expressions. Our modifications succeed in some cases, but is still unable to detect the real error source of some trickier cases.

In this report, we discuss the reasons why current error messages are flawless and misleading to novice programmers (Section 2); what an effective error message for beginners should convey and how to incorporate new designs into

Author's address: Scarlett Xu, wenwen.xu2@mail.mcgill.ca.

current type checking system (Section 3); some tricky cases to address (Section 4); related work in improving typing errors (Section 5) and possible ways to evaluate the effectiveness of improved messages (Section 6).

2 LIMITATIONS OF CURRENT TYPE ERROR MESSAGES

In this section, we discuss some specific features of OCaml that level up the threshold for beginners to understand what error messages convey. We present corresponding examples and analyze how the type check works and why it fails to give precise error messages.

2.1 Expression-based Constraints

Hindley/Milner type inference is one of the core algorithms that makes the OCaml language and other functional languages. It is fundamentally based on traversing the source code to collect a system of constraints, then solving that system to determine the types[Hindley 1969]. Different from some weakly typed languages like Java, some basic expressions have more constraints in OCaml. For example, a common mistake regarding conditional expressions happens when students forget to write the else branch. In this case, the then branch must have the type of `unit`. However, the concept about unit is strange to most novice programmers and the error message offers no concrete fixing suggestion as shown in the below example.

```
# let f lst = if lst = [] then lst;;
Error: This expression has type 'a list but an expression was expected of type unit
```

2.2 Left-to-Right Bias

Left-to-right bias is a side-effect of the unification algorithm in the Hindley/Milner type system that stops immediately when it fails to unify a constraint[Milner 1978][Hindley 1969]. Therefore, as the parser parses all tokens and builds up the AST from left to right, the typechecker always stops at the first error.

For instance, to typecheck a conditional expression, the OCaml compiler first typechecks the then branch, and then it typechecks the else branch. If two branches have different types, an error always gets raised inside the else branch during unification. This process thereby introduces a significant left-to-right bias: errors are always reported in the else branch. However, expression in then branch is equally likely to be the reason causing the type error, as in the following example:

```
# let f n = if n = 0 then n-1 else [];;
Error: This expression has type 'a list but an expression was expected of type int
```

Since the type checker automatically guesses types, it is difficult to tell which type should be returned exactly in both branches without type annotations or indication from function names. The type checker first infers the type of `n-1` as int and defines int as the return type. Hence, the type of expression in the else branch should also be int but the provided empty 'a list gives rise to the type conflict here. Additionally, left-to-right bias is prevalent in other expressions, such as pattern matching and application.

2.3 Type-sensitive Operators

In some languages, operators have broader usage than those in OCaml. For example, the operator `'+'` can be used in integer number addition, floating point number addition and even concatenation of strings in python. By contrast, operators in OCaml are viewed as a function with definite types. You have to use `'+'` for integer number additions, while floating point number addition requires `'+.'` and concatenation of two strings needs `'^'`. Therefore, the type checker cannot really distinguish whether the type error results from the operator itself or the arguments it takes. For example:

```
# let f x = (x + 1) *. x;;
Error: This expression has type int but an expression was expected of type float
```

In the AST of above function, the main operator `'*.'` is the parent of two sub-expressions. Therefore it imposes its restrictions on the types of `x + 1` and `x`, so `x+1` is expected to be a float. However, the inferred type of `x+1` is int actually.

There are two methods to remove the type error in **f**: (1) `let f x = (x +. 1) *. x;;` The function has type `float -> float` (2)`let f x = (x + 1) * x;;` The function has type `int -> int`. Both choices are well-typed, but OCaml type checker cannot tell the intention of programmers, therefore, a more effective kind of message structure is in need for display the acceptable types of an operator.

2.4 Syntax Error

Common syntax errors include omitting parenthesis or semicolons, however, such syntax errors often trigger type error messages instead of syntax error messages in OCaml. Although syntax errors are relatively easier to be spotted and fixed, what the error message conveys does not instruct the programmer to make correct fixing in most cases. For example:

```
# let l = [1;2;3 4];;
Error: This expression has type int
       This is not a function; it cannot be applied.
```

The compiler treats `3 4` as an application expression, where the int 3 is expected as a function with type `int -> int` because int 4 is viewed as its argument and the output should match the type of previous elements in this list.

According to Wu and Chen [Wu and Chen 2017], more than 45% of type errors were fixed by adding or removing pairs of parentheses or brackets. Missing parenthesis demands a non-leaf change of an AST, however, students are often unaware of the high frequency of causing type errors by wrong pairs of parentheses or brackets, or they were misled by the error messages when type errors were non-leaf. The following example illustrates this case.

```
# let f n = n-1 ;;
# f -1;;
Error: This expression has type int -> int but an expression was expected of type int
```

In this example, the argument that **f** should take is negative one, however, without proper parenthesis, the type checker take precedence of the operator - as the argument of **f**, which triggers the type conflict. What is valuable to novice programmers should be a more direct warning about parenthesis information.

3 IMPLEMENTATION OF IMPROVED ERROR MESSAGES

Ideally, effective error messages should be able to locate the source of errors precisely and give straightforward and informative enough explanations. The AST of the source code is constructed by nodes of expressions and different kinds of expressions are basic patterns to be analyzed during unification and error catching. Reasonably, it would be easier to implement the improved ones based on the current type system. Moreover, from novice programmers' perspectives, learning a new programming language usually starts from learning basic expressions. Therefore, we decide to design tailored diagnosis towards different kinds of expression(see Table 1) for the beginners.

At the current stage, all of the modifications happen in the typing functionalities in the OCaml compiler. The main work is to modify the typing of an expression with an expected type so that it can provide better error messages and allow controlled propagation of return type information. Then in each expression, we can conduct independent type checking on subexpressions, decomposing the type of a function to obtain argument types and quantities and passing newly designed error messages. In case of a unification failure, we report the types of the branches as they were computed before we tried to unify them.

In Section 3.1 and 3.2, we will present comparisons between the original and the new type error messages(marked with an asterisk) concerned with some ill-typed conditional and application expressions.

3.1 Application Expression

In an application expression, if the applied arguments have incorrect types and too many arguments are provided at the same time, the compiler reports the latter type error first. For example,

```
# let f = fun x y -> if x =0.0 then x *. 1.1 else y *. 1.1 in f 1.0 2 3.0 ;;
Error: This function has type float -> float -> float
       It is applied to too many arguments; maybe you forgot a `;'.
*Error:The function ‘f’ is applied to too many argument.
       It should take 2 arguments, but you provide 3 arguments.
```

The new message reminds programmers of the specific number of arguments **f** should take and the number of actual arguments provided. This is achieved by decomposing the applied function to get the total number of input arguments it has and counting the number of arguments practically applying to it. Hence, they have a clearer instruction on how many arguments should be removed rather than come up with a solution to add a semicolon somewhere indicated by the original message.

3.2 Conditional Expression

There are three kinds of type errors concerned with a conditional expression. To tackle the case without an else branch, we give a direct fixing suggestion and explain that the then branch of a conditional expression without an else branch is expected as type unit. For example:

```
# let f lst = if lst = [] then lst;;
Error: This expression has type 'a list but an expression was expected of type unit
*Error: Please add an else branch! The then branch of a conditional expression without an else branch,
        so it should have type unit but it has type int.
```

As the expression in else branch is to blame all the time due to the left-to-right bias, we decide to display both types of two branches by type checking them independently. The new error message is fair to show the differences between the two branches, as in the following example:

```
# let f n = if n = 0 then n-1 else [];;
Error: This expression has type 'a list but an expression was expected of type int
*Error: Two branches in a conditional expression should have the same type,
        but your then branch has type int and the else branch has type 'a list.
```

Table 1. General Categories of Type Errors

Main Categories	Sub categories
Application	Number of arguments mismatch; Types of arguments mismatch
Conditional	No else branch; Not a bool condition; Types of two branches mismatch
Pattern Matching	Patterns mismatch; Types of branches mismatch
Basic Syntax	Missing key word "rec"; Brackets Mismatch; Missing dereference symbol "!", etc.

At last, it is imperative to ensure that the condition has type `bool`. If not, we notify this in the new error message.

```
# let f n = if 1 then n-1 else n+1;;
Error: This expression has type int but an expression was expected of type bool
*Error: This expression is the condition of a conditional expression, so it should have type bool
        but it has type 'a list.
```

4 UNSOLVED PROBLEMS

As typechecking stops at the first type conflict, it is impossible to make use of the future environments to find the exact expression that triggers a type error. We can see such scenario in this case:

```
# let f = fun x y -> if x > 0 then x *. 1.1 else y *. 1.1;;
Error: This expression has type int but an expression was expected of type float
```

In such conditional expression, it is most likely to change `x>0` into `x>.0` to solve the type error because the return return types of two branches are the same if it has not added `x` as an int to the constraints. However, the error message concerned with `x` is reported in the then branch is misleading .

Still, in the conditional expression, typechecking two branches independently can return the same type, however, the free variables in two branches may have conflicting constraints on their types. A simple example illustrates this problem:

```
# let f a b = if a then print_int b else print_float b;;
Error: This expression has type int but an expression was expected of type float
```

Both `print_int b` and `print_float b` return type of `unit` during independent typechecking, however, the type of variable `b` has to be consistent in function **f**. Therefore, `b` in else branch is to blame as it has already been constraint to `int` in then branch.

Application with too many arguments leads to the type error all the time. In contrast, partial evaluation (i.e. application with fewer arguments) passes the type checker easily, or it invokes a warning message as this example shows:

```
# let _ = print_int 3; print_newline; print_int 3;;
Warning 5: this function application is partial,
maybe some arguments are missing.
33- : unit = ()
```

5 RELATED WORK

In order to provide better error messages and locate the source of type errors more precisely based on current type systems, many researchers have developed useful type debugging tools and algorithms.

- `Interactive debugging`: Tsushima and Asai developed an interactive OCaml type debugger to relocate the source of type errors, which asks a serial of questions about whether some expression has the same type as we intended[Ishii and Asai 2014]. By continuous narrowing down the scope of erroneous expression, it will make a final decision on which expression is the source of the type error. Moreover, by inheriting Chitil's[Chitil 2001] most general type tree theory which infers types compositionally, they also overcome the problem of standard type inference tree where a type of an expression can depend on or be constraint by the types of other expressions but cannot independently be inferred with its most general type[Tsushima and Asai 2012]. In addition, there is an interactive tool called NANOMALY[Seidel et al. 2016], which takes an opposite angle to to illustrate how an ill-typed program goes wrong step by step by generateing counterexample witnesses dynamically. By selecting a value as an input of the ill-typed function , NANOMALY shows the evaluating reduction steps until the problematic value triggers the type conflicts and ceases the evaluating process.
- `Concrete Explanation of Type Errors`:
  As left-to-right bias is a major problem of current type error messages, Charguéraud presents a modification to the traditional ML type inference algorithm that is able to remove nearly all of the left-to-right bias[Charguéraud 2015]. However, there remain a few cases where unification leads to side effects across subexpressions. Moreover, a framework called MYCROFT[Loncaric et al. 2016], which enables adoption of correcting set-based error explanation by retrofitting constraint-based type inference algorithms. When MYCROFT terminates and produces a correcting set, it has the advantage of constructing a human-readable error report.
  In the case where there are multiple errors, it is inconvenient for the programmer to keep recompile their code to get explanations for each error. There is a novel type inference algorithm[Kustanto and Kameyama 2010] to identify multiple errors in the form of error slices. They modify the standard unification algorithm so that it always terminates and returns a set of substitutions and an error set. This algorithm effectively breaks through the constraints in the traditional type inference system which stops at the first spot of type conflicts. For not only beginners but also experienced developers, such information about the existence of other errors is useful to fix one particular error and fixing multiple errors at one time also can greatly enhance debugging efficiency.
- `Analysis of type error fixing behaviors`: Marceau et al.[Marceau et al. 2011a] present a rubric which is able to identify error messages that students respond to poorly and assess the performance of various classes of error messages. They also dig deeper in understand the effectiveness of error messages[Marceau et al. 2011b] by holding interviews with students to know how they interpreted the errors as they worked on the problem. One of their observations emphasizes that the wording of error messages strongly influence students' debugging responses. For instance, some technical terms prevent them from understanding the error and even drag students attention to eliminate such warning but fail to make the actual fix. Additionally, Wu and Chen[Wu and Chen 2017] did a large scale study on the causes of type errors and made comparisons of the effectiveness of current error message delivery techniques in practice. Since their work is mostly with respect to the statistics analysis, they offer valuable summaries about main difficulty reasons that students struggled when debugging type errors: wrong type annotation, multiple library functions, wrong function composition, point-free function definition, and pattern matching. They also give valuable suggestions about designing wording of error messages and propose some features for future researchers to explore. For example, they find out providing instructions about changing some expression into another is more concrete than just mentioning type conflicts in error messages.

6 FUTURE WORK

There still remains work to be done. In the future, we plan to work on several fronts to incorporate new ideas into Learn-OCaml. Currently, we have covered only two categories of expressions in the OCaml compiler locally. Therefore, in order to make it more accessible to students, the next stage of this project should involve implementations on remaining cases and transplanting the *typecore* library to where Learn-OCaml compile source code.

The next goal of this project is to evaluate whether our improved error messages are indeed more effective than original ones for beginners. Also in the long run, we are interested in investigating more about how error fixing evolve as students getting more familiar with this language.

- `Logging Functionality`: In order to obtain enough data for evaluation, we need to implement a logging functionality on Learn-OCaml platform to record the following data points from each student:
  - 1.The number of steps to fix an error (i.e. how mant times students hit the **compile** button)
  - 2.The time duration between the occurrence of an error and its fixing.
  - 3.Every edit change from the ill-typed program to its well-typed version.
  - 4.Every error messages along the time steps.
- `Evaluation`: From the collected edits and error messages, one key measurement is to see whether the message tells the real reason and points to the right location. We should also assess the validity based on students' fixing strategies. Therefore, we need to distinguish whether they take action to fix the error or they just raise some exception to pass type checker.

  With the recorded debugging duration and compile times, we can conclude whether students with improved error feedback tend to spend less time and fewer steps on debugging type errors. Moreover, we can get the frequency distribution of each kind of type errors based on collected error messages to see whether there exists a difficulty hierarchy for various kinds of errors. For example, the type errors in a conditional expression or basic syntax errors cost less debugging time than missing parenthesis or wrong type annotations.
- `More Feature Designs`: Similar to a weekly screen usage report, a regular type error debugging report generated from the logging statistics can help students reflect on the most dominant and most time-consuming kinds of type error. It can periodically remind students of their coding performance so that they can be more careful and efficient when dealing with similar expressions next time. Moreover, although students learn from trials and errors, providing the difference between their answer and model solution dynamically can stop students from complicating simple problems and using unnecessary data structures. This idea could be achieved via Myers algorithm [Myers and Miller 1988], from which a shortest edit script can be generated to illustrate how to convert students' answers to model solutions.

In the future, it would be interesting to see if these ideas could be ported to Learn-OCaml so that students can have more juicy feedback on their code.

7 CONCLUSION

Although we only managed to improve type error messages for two kinds of expressions this semester, it lays the foundation for future modifications on remaining expressions. Inspired by Learn-OCaml, we also propose a feasible logging function to record user data and evaluate the effectiveness of improved error messages for students in COMP302.

ACKNOWLEDGMENTS

I am very grateful to my supervisor Prof.Pientka for her inspirational and valuable advice and feedback about this project across the term. In addition, Aliya Hameer provides me generous help on some knowledge of the OCaml compiler and details of the Learn-OCaml platform.

REFERENCES

Arthur Charguéraud. 2015. Improving Type Error Messages in OCaml. *Electronic Proceedings in Theoretical Computer Science* 198 (12 2015). https://doi.org/10.4204/EPTCS.198.4

Olaf Chitil. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 193–204.

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60.

Yuki Ishii and Kenichi Asai. 2014. Report on a User Test and Extension of a Type Debugger for Novice Programmers. *Electronic Proceedings in Theoretical Computer Science* 170 (12 2014). https://doi.org/10.4204/EPTCS.170.1

Cynthia Kustanto and Yukiyoshi Kameyama. 2010. Improving Error Messages in Type System. *IPSJ Online Transactions* 3 (2010), 125–138. https://doi.org/10.2197/ipsjtrans.3.125

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. *SIGPLAN Not.* 51, 10 (Oct. 2016), 781–799. https://doi.org/10.1145/3022671.2983994

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011a. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011b. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

Eugene W Myers and Webb Miller. 1988. Optimal alignments in linear space. *Bioinformatics* 4, 1 (1988), 11–17.

Eric Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). 228–242. https://doi.org/10.1145/2951913.2951915

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

Baijun Wu and Sheng Chen. 2017. How type errors were fixed and what students did? *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 105.