Improving Type Error Messages in OCaml

Arthur Charguéraud

Inria
& LRI, Université Paris Sud, CNRS
arthur.chargueraud@inria.fr

Abstract

Cryptic type error messages are a major obstacle to learning OCaml. In many cases, error messages cannot be interpreted without a sufficiently-precise model of the type inference algorithm. However, improving type error messages in ML is a hard problem. This problem has received quite a bit of attention over the past two decades, and many different strategies have been considered. Unfortunately, none of these strategies has so far proved its ability to scale up to a full-blown ML implementation.

Perhaps we should turn the problem around: given that we need a typing algorithm that can be implemented within the OCaml compiler and that scales up to the full-blown language, what can be done to improve error messages? In this work, we show that, with just a few hundred lines of code, we can integrate a fallback typing algorithm producing more helpful messages. Our work is implemented in the form of a patch to the OCaml compiler. This patch should benefit not just to beginners, but also to experienced programs developing large-scale OCaml programs.

1. Introduction

Will the OCaml community remain stuck forever with error messages that make many potential adopters of OCaml run away before they really get a chance to appreciate the language?

We believe that OCaml really deserves better parsing and typing error messages. In this work, we focus on typing errors. We show that, with a small number of carefully-crafted changes to the order in which unifications are performed by the type inference engine, we are able to generate messages that, we argue, provide the user with much more useful information for locating and fixing the error.

The main features of our approach are:

– improved error messages for function applications; in particular, better treatment of errors involving higher-order functions (e.g., `List.fold`), and arithmetic operators (e.g., '`+`' in place of '`+.`', or '`-1`' not surrounded by parentheses);

– improved error messages for subexpressions that do not have the type expected by the language construction in which they appear, e.g., a while loop condition that does not have type `bool`;

– improved error messages for incompatible branches in conditional and pattern matching constructs;

– improved error messages for missing '`()`' argument, missing '`!`' operator, and missing '`rec`' keyword.

In Section 2, we present these features in reverse order, so as to start with the simpler ones. In each case, we explain in a few lines the idea of the algorithm, and then illustrate its application on concrete examples. In particular, these examples cover error patterns that arises frequently in code written by OCaml beginners.

Our approach is implemented as a patch to the OCaml compiler. The patched compiler runs exactly like the original compiler, excepts when it encounters a top-level definition that fails to type-check. If the typing of the top-level definition concerned involves GADTs, then we simply display the traditional error message (because our current implementation does not yet support improved messages for errors involving GADTs). Otherwise, we type-check again the same top-level definition using our modified algorithm, and report a possibly-different error message.

Thanks to this strategy, our approach scales up to large and complex programs. For example, we tested the ability of our patched compiler to report improved typing error messages for errors artificially introduced in functions of several hundred lines located in the middle of the implementation of the OCaml type-checker itself.

2. Algorithm

Message for missing '`()`'. When reaching a unification error between a type of the form "`unit -> t`" and a type `u` that does not unify with an arrow type, we add to the error message the sentence: "You probably forgot to provide '`()`' as argument somewhere."

Example 1 from the appendix shows a program containing a call to `read_int` that is missing its argument. The code is followed by error message produced by the original OCaml compiler, then that produced by our patched compiler. (Note that the main body of the error message, explaining the failure to type-check the application, is also modified; we will explain this change later, when discussing typing of applications.)

Example 2 shows another instance of a missing unit argument, this time on a call to `print_newline`. Here, the main body of the error corresponds to the strict-sequence check, which we activate by default.[1] Note that, in this particular case, we remove the word "somewhere" because we know that the location reported with the error is the appropriate one.

Message for missing '`!`'. When reaching a unification error between a type of the form "`t ref`" and a type `u` that does not unify with it, we add to the error message the sentence: "You probably forgot a '`!`' operator somewhere."

Example 3 illustrates a call to "`print_int r`", where `r` is a reference. Example 4 shows that it is important to leave the word "somewhere" in the message, because if the expression of type "`int ref`" is the result of a function call, then the type-checker cannot guess whether the '`!`' operator is missing in the function definition or at the call site.

Message for missing '`rec`'. When traversing non-recursive let-bindings, we add the bound names as ghost names to the environ-

[1] OCaml with strict-sequence flag activated gives the message: "This expression has type `unit -> unit` but an expression was expected of type `unit`." However, beginners, when using `print_newline` in their first OCaml program, do not yet know anything about the arrow type.

ment, so that if we later obtain an "unbound value" error, we are able to test if the variable name would have been in scope if it had been bound by a `let rec` instead of being bound by a simple `let`; if so, we add the message: "You are probably missing the 'rec' keyword on line n.". Example 5 illustrates such a scenario.

Message for subexpressions of language constructs. If a language construct expects a subexpression to be of a given type (e.g., loop conditions should have type `bool`) but the subexpression does not have this type, then we produce a specific error-message, e.g., "This expression is the condition of a while loop, so it should have type `bool`, but it has type `foo`."

Example 6 provides an example for the case of a while loop condition. Example 7 provides an example of a conditional missing its "else" branch. By the way, observe the amazing error message produced by the original OCaml type-checker: "The constructor :: does not belong to type unit."

Message for ill-typed applications. To type-check a conditional or a pattern matching, we first type-check each of the branches independently (at least, reasonably independently, as explained further), then we unify the types of the branches one by one, and at the very end we unify the type of the branches with the expected type of the expression (if any). In case of a unification failure, we report the types of the branches as they were computed before we tried to unify them.[2]

The goal here is to limit the effects of the left-to-right bias, which is significant when we unify the branches one by one in order. In particular, we want to avoid focusing the programmer's attention to one particular branch while the error may very well be located in a previous branch. Example 8 shows a conditional whose branches return values of types `int` and `float`. Example 9 shows a similar situation in a pattern matching. Example 10 presents a more subtle program that illustrates how, with the traditional type-checker, information gets propagated across the branches. Our modified algorithm avoids this kind of propagation.

Example 11 shows that our approach nevertheless allows for some amount of side-effects between the typing of the different branches. The variable `x` involved has no fixed type when starting to type-check the pattern-matching. Its type gets unified with `int` in the first branch, and therefore fails to unify with `float` in the second branch. We believe that this remaining left-to-right bias is less of a concern, because it typically only involves a variable, which is blamed by the error message. So, if the programmer provides a type annotation for the variable that does not get assigned the expected type, she should obtain in the next error the precise location of the source of the problem.

Message for incompatible branches. To type-check an application, we compute the most-general type of the function and of each of the arguments provided, independently. Then, we try to unify the types. If this unification process fails, we locate the error on the entire application, and we report a message of the form: "The function 'foo' expects N arguments of type 'bla' and 'bla', but it is given M arguments of type 'bla' and 'bla' and 'bla'.",

Here again, the goal is to limit the effects of the left-to-right bias, and to avoid the programmer focusing its attention on one particular argument that may not be the one to blame. We argue for the benefits of the new error messages through several examples. Example 12 illustrates the use of '`+`' in place of '`+.`'. The new message makes it clear that '`+`' operates on `int` and not on `float`. Example 13 illustrates a call of a function on '`-1`' not surrounded by parenthesis. The new message makes it clear that '`-`' is treated as a binary operator and not a unary one.

Example 14 illustrates how the new error message helps detecting swapped arguments in a call to `Array.fold_left`. Example 15 shows how the new error message helps detecting swapped arguments in the function passed as argument to `Array.fold_left`. By the way, note the cryptic message produced by the original type-checker: "The type variable 'a occurs inside 'a list". Example 16 presents a type error on a call to `fold_left`, due to a mismatch between the type of the function and the type of the list provided. Last but not least, Example 17 illustrates that our implementation smoothly extends to optional arguments.

Implementation. Our implementation is relatively lightweight. We needed only 6 lines of code to improve messages for '`()`', also 6 lines for '`!`', 20 lines for missing '`rec`'. For incompatible branches, we needed 50 new lines of code, plus minor patching of 100 existing lines of code. For applications, we needed 100 new lines of code, plus minor patching of 200 existing lines of code.

3. Related work

There is a large literature on the production of better typing errors for ML, including recent publications in conferences such as ICFP and POPL, and including rather unexpected method —e.g., the use of search procedures for finding similar programs that do type-check [4]. Heeren's PhD thesis [2], entitled "Top Quality Type Error Messages", summarizes the main possible directions to producing good error messages: (1) tracing everything that contributes to the error; this approach, however, leads to very verbose messages; (2) attempting to infer the most likely cause of the error; this approach, however, is problematic when it fails to guess right, (3) modifying the unification algorithm; this approach, however, requires finding a new algorithm.

Regarding the third direction, several researchers have argued that a very central ingredient for achieving intuitive error messages is the elimination of the left-to-right bias associated the way unifications are traditionally performed. To avoid the left-to-right bias, Bernstein and Stark [1], and Yang [6] propose to type subterms bottom-up, returning a type for the term and all of its free variables, and to then try to unify the types of the free variables. However, such approaches that involve typing fully-open terms tend to be fairly impractical. McAdam [5] proposes a technique that is able to eliminate all left-to-right bias without going as far as typing fully-open terms. McAdam's approach consists of typing subterms bottom-up, returning a type and a substitution for flexible type variables, and then trying to unify the substitutions involved.

In our work, we also type subterms bottom-up, and these subterms are processed relatively independently. We thereby remove most of the left-to-right bias, even though, as illustrated by Example 11, there remains a few cases where unification leads to side-effects visible between subterms. We find our approach to be a good compromise between the aim to treat subterms independently, and the need for simplicity and efficiency. Also, our algorithm can be implemented easily, reusing all the data structures and infrastructure already existing in the OCaml compiler.

The possibility of performing unifications in different orders is also discussed in by Li and Yi [3], who show how several existing algorithms ($\mathcal{W}$, SML/NJ's algorithm, OCaml's algorithm, $\mathcal{H}$, $\mathcal{M}$) can be viewed as particular instantiations of a general algorithm. However, Li and Yi do not discuss which instantiations might be best suited for error reporting, and do not discuss the treatment of n-ary applications and pattern matching.

[2] Remark: in order to properly report error messages, we need to save copy of the types of the branches before starting to unify these types. We implement those copy by computing the type scheme associated with the types involved before unifying them. This generalization operation is efficiently implemented; it is basically equivalent to naming each of the arguments using a let-binding before constructing the application.

4. Conclusion

Our work shows that, with reasonable effort, we are able to significantly improve type error messages generated by the OCaml compiler. Our approach integrates within the full-blown language, and scales up to large-scale programs with virtually no overhead. While our work has been mainly motivated by OCaml, we believe that our approach to reporting errors for applications and branching constructs could be similarly applied in other functional programming languages, such as SML, Haskell, or Coq.

References

[1] Karen L. Bernstein and Eugene W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, November 08 1995.

[2] Bastiaan Heeren et al. *Top quality type error messages*. Utrecht University, 2005.

[3] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

[4] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434. ACM, 2007.

[5] Bruce J. McAdam. On the unification of substitutions in type inference. In Kevin Hammond, Anthony J. T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL'98)*, volume 1595 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, September 1998.

[6] Jun Yang. Explaining type errors by finding the source of a type conflict. In Philip W. Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Scottish Functional Programming Workshop*, volume 1 of *Trends in Functional Programming*, pages 59–67. Intellect, 1999.

A. Appendix

The remaining of the paper contains the examples described in Section 2. All these examples have been processed by our patched compiler, which is publicly available and can be tested using the following commands.

```
git clone https://github.com/charguer/ocaml.git
cd ocaml
./configure && make world.opt
./ocamlc.opt -I stdlib -easy foo.ml
```

```
***************************** Example 1 *********************************
let x = read_int in   (* missing unit argument *)
print_int x
------------------------------ old error --------------------------------
File "examples/example_missing_unit_readint.ml", line 2, characters 10-11:
Error: This expression has type unit -> int
       but an expression was expected of type int.
------------------------------ new error --------------------------------
File "examples/example_missing_unit_readint.ml", line 2, characters 0-9:
Error: The function ‘print_int’ expects one argument of type [int],
       but it is given one argument of type [unit -> int].
You probably forgot to provide ‘()’ as argument somewhere.

***************************** Example 2 *********************************
let _ =
   print_int 3;
   print_newline;    (* missing unit argument *)
   print_int 5
------------------------------ old error --------------------------------
File "examples/example_missing_unit_newline.ml", line 3, characters 3-16:
Warning 5: this function application is partial,
maybe some arguments are missing.
------------------------------ new error --------------------------------
File "examples/example_missing_unit_newline.ml", line 3, characters 3-16:
Error: This expression is followed by a semi-column, so it should have type
       [unit] but it has type [unit -> unit].
You probably forgot to provide ‘()’ as argument.

***************************** Example 3 *********************************
let r = ref 1 in
print_int r       (* should be [!r] *)
------------------------------ old error --------------------------------
File "examples/example_ref_missing_bang.ml", line 2, characters 10-11:
Error: This expression has type int ref
       but an expression was expected of type int.
------------------------------ new error --------------------------------
File "examples/example_ref_missing_bang.ml", line 2, characters 0-9:
Error: The function ‘print_int’ expects one argument of type [int],
       but it is given one argument of type [int ref].
You probably forgot a ‘!’ operator somewhere.

***************************** Example 4 *********************************
let f x y =
  let z = ref 0 in
  z := !z + x;
  z := !z + y;
  z                 (* missing "!" here? *)
let _ =
  print_int (f 3 4) (* or maybe there? *)
------------------------------ old error --------------------------------
File "examples/example_missing_bang_delayed.ml", line 7, characters 12-19:
Error: This expression has type int ref
       but an expression was expected of type int.
------------------------------ new error --------------------------------
File "examples/example_missing_bang_delayed.ml", line 7, characters 2-11:
Error: The function ‘print_int’ expects one argument of type [int],
       but it is given one argument of type [int ref].
You probably forgot a ‘!’ operator somewhere.

***************************** Example 5 *********************************
let facto n =   (* missing [rec] *)
   if n = 0 then 1 else n * facto (n-1)
------------------------------ old error --------------------------------
File "examples/example_let_missing_rec.ml", line 2, characters 28-33:
Error: Unbound value facto
------------------------------ new error --------------------------------
File "examples/example_let_missing_rec.ml", line 2, characters 28-33:
Error: Unbound value facto.
You are probably missing the ‘rec’ keyword on line 1.

***************************** Example 6 *********************************
let _ =
   while 1 do () done
------------------------------ old error --------------------------------
File "examples/example_while_bad_condition.ml", line 3, characters 9-10:
Error: This expression has type int but an expression was expected of type
        bool.
------------------------------ new error --------------------------------
File "examples/example_while_bad_condition.ml", line 3, characters 9-10:
Error: This expression is the condition of a while loop,
        so it should have type [bool] but it has type [int].

***************************** Example 7 *********************************
let f x y =
   if x > y then [x]  (* missing "else [y]" *)
------------------------------ old error --------------------------------
File "examples/example_missing_else.ml", line 2, characters 18-20:
Error: This variant expression is expected to have type unit
       The constructor :: does not belong to type unit
------------------------------ new error --------------------------------
File "examples/example_missing_else.ml", line 2, characters 17-20:
Error: This expression is the result of a conditional with no else branch,
        so it should have type [unit] but it has type ['a list].
```

```
****************************** Example 8 *********************************
let f b =
  if b then 0 else 3.14 (* should have been 0. *)
------------------------------ old error --------------------------------
File "examples/example_incompatible_else.ml", line 2, characters 19-23:
Error: This expression has type float but an expression was expected of
       type
         int.
------------------------------ new error --------------------------------
File "examples/example_incompatible_else.ml", line 2, characters 2-23:
Error: The then-branch has type [int]
       but the else-branch has type
       [float].
       Cannot unify type [int] with type [float].

****************************** Example 9 *********************************
let headval = function      (* intended to be of type [int list -> float]
       *)
   | [] -> 0                (* intended [0.] instead of [0] *)
   | a::_ -> float_of_int a
------------------------------ old error --------------------------------
File "examples/example_match_incompat_branches.ml", line 3, characters
       13-27:
Error: This expression has type float but an expression was expected of
       type
         int.
------------------------------ new error --------------------------------
File "examples/example_match_incompat_branches.ml", line 3, characters
       13-27:
Error: the previous branches produce values of type [int]
       but this branch has type [float].

****************************** Example 10 ********************************
let rec sum = function
   | [] -> 0                (* error might be 0 instead of 0. *)
   | a::l -> a +. (sum l)   (* or it might be +. instead of + *)
------------------------------ old error --------------------------------
File "examples/example_match_incompat_branches_2.ml", line 3, characters
       18-25:
Error: This expression has type int but an expression was expected of type
         float.
------------------------------ new error --------------------------------
File "examples/example_match_incompat_branches_2.ml", line 3, characters
       13-25:
Error: the previous branches produce values of type [int]
       but this branch has type [float].

****************************** Example 11 ********************************
let f b x =
  if b
    then print_int x
    else print_float x
------------------------------ old error --------------------------------
File "examples/example_if_propagate.ml", line 5, characters 21-22:
Error: This expression has type int but an expression was expected of type
         float.
------------------------------ new error --------------------------------
File "examples/example_if_propagate.ml", line 5, characters 9-20:
Error: The function ‘print_float’ expects one argument of type [float],
       but it is given one argument of type [int].

****************************** Example 12 ********************************
let _ =
  print_float (2.0 + 3.0)     (* should be [+.] instead of [+] *)
------------------------------ old error --------------------------------
File "examples/example_add_bad.ml", line 2, characters 15-18:
Error: This expression has type float but an expression was expected of
       type
         int.
------------------------------ new error --------------------------------
File "examples/example_add_bad.ml", line 2, characters 19-20:
Error: The function ‘+’ expects 2 arguments of types [int] and [int],
       but it is given 2 arguments of types [float] and [float].

****************************** Example 13 ********************************
let _ =
   succ -1       (* missing parentheses around [-1] *)
------------------------------ old error --------------------------------
File "examples/example_f_minus_one.ml", line 2, characters 3-7:
Error: This expression has type int -> int
       but an expression was expected of type int.
------------------------------ new error --------------------------------
File "examples/example_f_minus_one.ml", line 2, characters 8-9:
Error: The function ‘-’ expects 2 arguments of types [int] and [int],
       but it is given 2 arguments of types [int -> int] and [int].

****************************** Example 14 ********************************
let _ = List.fold_left (fun acc x -> acc + x) [1;2;3] 0
(* above, last two arguments of fold_left were swapped *)
------------------------------ old error --------------------------------
File "examples/example_fold_left_swap_arg.ml", line 1, characters 46-53:
Error: This expression has type 'a list
       but an expression was expected of type int.
------------------------------ new error --------------------------------
File "examples/example_fold_left_swap_arg.ml", line 1, characters 8-22:
Error: The function ‘List.fold_left’ expects 3 arguments of types
       ['a -> 'b -> 'a] and ['a] and ['b list],
       but it is given 3 arguments of types [int -> int -> int]
        and [int list] and [int].

****************************** Example 15 ********************************
let rev_filter f l =
  List.fold_left (fun x acc -> if f x then x::acc else acc) [] [1; 2; 3]
(* swapped the parameters of the higher-order function *)
------------------------------ old error --------------------------------
File "examples/example_fold_left_swap_app_2.ml", line 2, characters 43-44:
Error: This expression has type 'a list
       but an expression was expected of type 'a.
       The type variable 'a occurs inside 'a list
------------------------------ new error --------------------------------
File "examples/example_fold_left_swap_app_2.ml", line 2, characters 2-16:
Error: The function ‘List.fold_left’ expects 3 arguments of types
       ['a -> 'b -> 'a] and ['a] and ['b list],
       but it is given 3 arguments of types ['c -> 'c list -> 'c list]
        and ['d list] and [int list].

****************************** Example 16 ********************************
let _ = List.map (fun x -> x + 1) [2.0; 3.0]
(* should have been [+.] instead of [+], or
   should have been [2;3] instead of [2.0;3.0] *)
------------------------------ old error --------------------------------
File "examples/example_map_bad.ml", line 1, characters 35-38:
Error: This expression has type float but an expression was expected of
       type
         int.
------------------------------ new error --------------------------------
File "examples/example_map_bad.ml", line 1, characters 8-16:
Error: The function ‘List.map’ expects 2 arguments of types ['a -> 'b]
        and ['a list], but it is given 2 arguments of types [int -> int]
        and [float list].

****************************** Example 17 ********************************
let f ?(x=true) y ?z ~t u =
  if x && t
     then ((match z with None -> [y] | Some x -> x)) @ u
     else [y]
let _ =
  f ~x:false 0 ~t:false [3.0]
------------------------------ old error --------------------------------
File "examples/example_apply_labels.ml", line 6, characters 25-28:
Error: This expression has type float but an expression was expected of
       type
         int.
------------------------------ new error --------------------------------
File "examples/example_apply_labels.ml", line 6, characters 2-3:
Error: The function ‘f’ expects 5 arguments of types ?x[bool option] and
       ['a] and ?z['a list option] and ~t[bool] and ['a list],
       but it is given 4 arguments of types ~x[bool] and [int] and ~t[bool]
        and [float list].
```