Enumerating Counter-Factual Type Error Messages with an Existing Type Checker (poster + demo)

Kanae Tsushima
Kyoto University, Japan
tsushima@fos.kuis.kyoto-u.ac.jp

Olaf Chitil
University of Kent, UK
O.Chitil@kent.ac.uk

The Hindley-Milner type system is a foundation for most statically typed functional programming languages, such as ML, OCaml and Haskell. This type system has many advantageous, but it does make type debugging hard: If a program is not well-typed, it can be difficult for the programmer to locate the cause of the type error, that is, to determine where to change the program how.

Many solutions to the problem have been proposed. Here we propose a new solution with two distinctive advantages: It is easy to use for the functional programmer, because it appears to be only a minor extension of the type error messages they are already familiar with. It is easy to implement, because it does not require the implementation of a new type checker, but instead reuses any existing one as a subroutine (like [2]).

Consider the following ill-typed OCaml program[1] and the type error message produced by the OCaml compiler:

```
let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

Error: This expression has type float
       but an expression was expected of type string
```

The message identifies the underlined expression `2.0` in the program as the location of the type error. The message gives two different types for the expression `2.0`: Its actual type and an expected type. The expected type is determined by the context of `2.0`, the rest of the program. As the expected type is different from the actual type, it is a counter-factual type [1]. The message basically says that if the expression `2.0` was replaced by some expression of the expected type, then this part of the program would be well-typed (there might be further type errors elsewhere). Indeed, replacing `2.0` by any string, for example `"2.0"`, produces a well-typed program.

So if the type error message identified the type error location correctly, then the message with its actual and expected type is very helpful. However, the subexpression `2.0` might be correct and the programmer might have confused the string concatenation operator `^` with the floating point exponentiation operator `**`. In that case a type error message like the following would have been helpful:

[1] Library function `List.map` applies its first argument, a function, to each element of its second argument, a list.

```
let f n lst = List.map (fun x -> x ^ n) lst in
f  2.0

Error: This expression has type string -> string -> string
       but an expression was expected of type 'a -> float -> 'b
```

In this work, we first show how to produce such counter-factual type error messages for *all* potential locations of type errors. Although we can construct example programs with many potential type error locations, we believe that in practice a program contains a large, well-typed part, which provides a context to limit the number of potential type error locations and yield informative counter-factual types. Nonetheless there are many potential type error locations.

The second part of our proposal is to apply algorithmic debugging to find the correct error location. The programmer only has to state whether an actual type agrees with their intentions. In the example session below the input of the user, *y*es or *n*o, is given in italics.

```
let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

1. Should this expression have type float? y

let f n lst = List.map (fun x -> x ^ n) lst in
f  2.0

2. Should this expression have type string -> string -> string? n

3. Type error located:
let f n lst = List.map (fun x -> x ^ n) lst in
f  2.0

This expression has type string -> string -> string
but an expression was expected of type 'a -> float -> 'b
```

After two questions the type debugger identifies the correct type error location. It gives the full counter-factual type error message so that the programmer can determine how to correct the error.

The main contribution of this work is that we present a method that uses a standard type checker to enumerate locations that potentially cause the type error. For each such location an actual and a counter-factual type are computed. Adding our method to existing compilers requires only limited effort but improves type error debugging substantially.

References

1. Chen, S., and M. Erwig. "Counter-Factual Typing for Debugging Type Errors," *Proceedings of the 41th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'14)*, (2014).
2. Tsushima, K., and K. Asai. "An Embedded Type Debugger," *Proceedings of the 24th International Workshop on Implementation of Functional Languages (IFL'12)*, pp. 190–206, Springer (2013).