Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker

Kanae Tsushima
National Institute of Informatics
Japan
k_tsushima@nii.ac.jp

Olaf Chitil
University of Kent
United Kingdom
oc@kent.ac.uk

Joanna Sharrad
University of Kent
United Kingdom
jks31@kent.ac.uk

ABSTRACT

The cause of a type error can be very difficult to find for the Hindley-Milner type system. Consequently many solutions have been proposed, but they are hardly used in practice. Here we propose a new solution that provides counter-factual type error messages; these messages state what types specific subexpressions in a program should have (in contrast to the types they actually have) to remove a type error. Such messages are easy-to-understand, because programers are already familiar with them. Furthermore, our solution is easy-to-implement, because it reuses an existing type checker as a subroutine. We transform an ill-typed program into a well-typed program with additional λ-bound variables. The types of these λ-bound variables yield actual and counter-factual type information. That type information plus intended types added as type annotations direct the search of the type debugger.

CCS CONCEPTS

• **Software and its engineering → Functional languages**.

KEYWORDS

Hindley-Milner type system, OCaml

ACM Reference Format:
Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. 2020. Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

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 benefits, but it does make type debugging hard: if a program is not well-typed, it is difficult for the programmer to locate the cause of the type error, that is, to determine where to change the program and how.

Many solutions to the problem have been proposed in the literature (see Section 8). Here we propose a new solution with two distinctive advantages: It is easy to understand 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.

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 it should be an expression 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. 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:

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

Error: This expression has type
    string -> string -> string
but it should be an expression of type
    'a -> float -> 'b
```

In this paper we first show how to produce such counter-factual type error messages for *all* potential locations of type errors. Although a program may contain many potential type error locations, we assume 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 which yields informative counter-factual types.

[1] Library function `List.map : ('a -> 'b) -> 'a list -> 'b list` applies its first argument, a function, to each element of its second argument, a list.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There are still too many potential type error locations for showing them all, embedded in counter-factual type error messages, to the programmer. Hence the second part of our proposal is to run an interactive debugger to find the correct error location. The programmer only has to state whether an actual type or expected type agrees with their intentions. Sometimes both types do not agree with the programmer's intention. In that case the programmer can also enter another type. In the example session below the input of the user is given in italics.

```
Interactive counter-factual type debugger.

1. Choose your intended type for this expression.

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

A: float
B: string
Your choice (C: another type): A

2. Choose your intended type for this expression.

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

A: string -> string -> string
B: 'a -> float -> 'b
Your choice (C: another type): B

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 it should be an expression of type
    'a -> float -> 'b
```

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

In this paper we make the following contributions:

- We define a new method for enumerating all counter-factual type error messages for an ill-typed program. The type checker of a real functional programming system is too complex to be easily replaced by something new. Hence we do not develop a new type checking algorithm, but instead describe an algorithm that reuses any existing type checker, which is treated as a black box (Sections 2 and 3).
- We provide an argument that it is sufficient to only consider leafs in the abstract syntax tree of the ill-typed program as potential type error locations (Section 2).
- We extend our method to produce counter-factual type error messages also for expressions that have to be polymorphic for the program to become well-typed (Section 4).
- The time complexity of our basic method is too high to be used for medium-sized or large programs. Hence we show how we can first execute a known type error slicing algorithm to then obtain counter-factual type error messages more quickly (Section 5).
- We define an interactive debugger that identifies the correct type error location building on our method for enumerating counter-factual type error messages (Section 6).
- We evaluate our prototype implementation with a number of ill-typed programs (Section 7).

2 IDEA: COUNTER-FACTUAL TYPES FROM AN EXISTING TYPE CHECKER

We reuse an existing type checker by giving it many variations of our ill-typed program to check. Viewing the type checker as a black box means that we expect the type checker to tell us only whether the program is well-typed or ill-typed. If the program is well-typed, then the type checker shall also tell us the type of the given program. If the program is ill-typed, then we demand no further information. In particular we do not use any details of the type checker's own error message(s).

2.1 Only Leaves as Potential Error Locations

In all our previous examples potential type error locations were simple variables or constants, not more complex expressions. In other words, a location was a leaf of the abstract syntax tree, never an inner node of the abstract syntax tree.

Our method works for both leaves and inner nodes of an abstract syntax tree, but here we argue that to debug all type errors it is sufficient to consider only leaf locations.

Consider the following ill-typed example:

```
(fun (x, y) -> if x then y else y + 1) [true; 1]
```

We assume that the programmer intended the argument to be the tuple `(true, 1)`. So arguably the whole argument `[true; 1]` is the type error location. However, because we consider only leaves in the abstract syntax tree as potential type error locations, we will instead offer the list constructor itself as type error location and produce the message

```
(fun (x, y) -> if x then y else y + 1) [true; 1]

Error: This expression has type
    'a -> 'a -> 'a list
but it should be an expression of type
       bool -> int -> bool * int
```

This message states the error more indirectly. The message should make the programmer consider replacing `[..;..]` by the function `fun x -> fun y -> (x,y)`. Subsequently simplifying `(fun x -> fun y -> (x,y)) true 1` to the expression `(true,1)` should be trivial.

In the worst case a leaf-location message may suggest to substitute a complex function that rearranges a whole subtree of the abstract syntax tree. Here "rearrange" is the keyword, because the arguments are usually not superfluous but needed. Limiting ourselves to leaves in the syntax tree has the advantage that the expression identified as erroneous is always simple.

In general we can limit ourselves to leaves as potential type error locations, because any program without any leaves[2] is well-typed. Only the use occurrences of variables, data constructors and constants lead to type constraints that cause type errors.

2.2 Single- and Multiple-Location Type Errors

Consider the ill-typed expression `[1;2;3.]`, which mixes integers and floating point numbers in a list. Two different single locations could be the cause of ill-typedness:

- The subexpression `3.` may be the type error location. Replacing it by an integer constant such as `3` would yield the well-typed expression `[1;2;3]`.
- The list constructor `[..;..;..]` (cf. next subsection) may be the type error location. Replacing it by, for example, a tuple `(..,..,..)` would yield the well-typed expression `(1,2,3.)`.

We call both *single-location type errors*.

Another way to correct the ill-typed expression would be to simultaneously replace `1` by `1.` and `2` by `2.`, yielding the well-typed expression `[1.;2.;3.]`. We call this a *multiple-location type error*, more precisely a *2-location type error*.

In general, a k-location type error states that replacing the expressions at k distinct locations in the program makes the program well-typed.

In practice, ill-typed programs often contain many independent type errors. However, we are not aware of any agreed definition of when two type errors are independent and when they are not. Our definition of k-location type error handles programs with many type errors, no matter whether there exist dependencies amongst these k locations or not.

Our method can handle k-location type errors for any k. However, to simplify the discussion, we will in most of this paper only consider searching for a single location that causes a program to be ill-typed. In practice, even when we enumerate all type errors, it is sensible to first enumerate all single location type errors, then 2-location type errors, etc. It is very likely that the programmer will find the actual cause of ill-typedness early in this list and thus does not have to look at errors that include numerous locations.

2.3 Program Syntax and the Syntax Tree

In the previous example we considered `[..;..;..]` as an atomic syntactic construct, which has the type `'a -> 'a -> 'a -> 'a list` and semantically denotes a function constructing a list from three arguments (`fun x -> fun y -> fun z -> [x; y; z]`). Desugaring this list construction into uses of the list constructor `(::)` and the empty list is not desirable, because desugared expressions in error messages would be confusing for the programmer. Similarly `if..then..else..` is an atom of type `bool -> 'a -> 'a -> 'a`.

In general we treat most syntactic constructs of a programming language as constants with fixed types. In the syntax tree of a program they are leafs.

[2]More precisely: any program where we replaced every leaf by a different fresh variable.

2.4 Determining Potential Error Locations with Expected Types

Let us consider the simple program `1.0 + 2.0`. Because the operator `+` demands integer numbers but `1.0` and `2.0` are floating point numbers, the program is ill-typed. Internally the program is represented as an application of the operator `+` to two arguments, that is, `(@ (+) 1.0 2.0)`.

The program has three leaf locations. We investigate for each leaf whether it is a potential single type error location by type checking a corresponding variant of our program:

(1) `fun hole -> @ hole 1.0 2.0`
(2) `fun hole -> @ (+) hole 2.0`
(3) `fun hole -> @ (+) 1.0 hole`

So we replaced a potential type error location by a new variable `hole` and added a λ-binding for the variable to the whole program. The λ-binding ensures that the program has no free variables and allows us to obtain a type for `hole` from the type checker.

We run the existing type checker on each of the three program variants. Programs (2) and (3) are ill-typed. Hence replacing the variable `hole` by any expression of any type would not make the program well-typed. Consequently the locations `1.0` and `2.0` are not potential single type error locations.

Program (1) is well-typed and its inferred type is `(float -> float -> 'a) -> 'a`. So the type of the variable `hole` is `float -> float -> 'a`. Consequently `(+)` is a potential type error location and we can produce the following message:

```
1. + 2.

Here should be an expression of type
    float -> float -> 'a
```

2.5 Obtaining Actual Types Too

The error message above does not yet include the actual type of `+`. We can also obtain that type if we do not simply replace a potential type error location by a variable `hole`, but instead apply a variable `hole` to the potential type error location. Again we λ-bind the variable `hole`. So instead of the 3 program variants listed before, we type check the following 3 variants:

(1) `fun hole -> @ (hole (+)) 1.0 2.0`
(2) `fun hole -> @ (+) (hole 1.0) 2.0`
(3) `fun hole -> @ (+) 1.0 (hole 2.0)`

As before, only variant (1) is well-typed. For this variant the inferred type is `((int -> int -> int) -> (float -> float -> 'a)) -> 'a`. So the type of the variable `hole` is `(int -> int -> int) -> (float -> float -> 'a)`, which contains both the actual and the expected type of the potential type error location. Thus we can produce the complete message:

```
1. + 2.

Error: This expression has type
    int -> int -> int
but it should be an expression of type
    float -> float -> 'a
```

Naturally we could have obtained the actual type of `+` by just type checking the program `(+)`. However, in general a potential type

error location may not be a predefined function or data constructor, but some variable that is λ- or let-bound in the ill-typed program. Our method of applying a variable `hole` to the potential type error location works in all situations.

3 OBTAINING COUNTER-FACTUAL TYPES FOR THE SIMPLY-TYPED λ-CALCULUS

In this section we formalise our idea for a small core functional language, the simply-typed lambda calculus $\lambda^{\rightarrow}$. Whereas in the preceding section we focussed on 1-location type errors, we now consider the general case of k-location counter-factual type errors. Recall that a k-location counter-factual type error states that replacing the expressions at k distinct locations in the program by expressions of different, expected types, makes the program well-typed.

The syntax and types of $\lambda^{\rightarrow}$ are shown in Figure 1. The constants include numeric literals, tuple data constructors and list data constructors. Each constant and each variable has location information, l, which uniquely identifies each occurrence in the program.

Figure 2 gives the algorithm for obtaining counter-factual type errors for $\lambda^{\rightarrow}$. The capitalised type-writer font functions (`GENVAR` etc.) are external.

The last function, *get_cft_nloc*, obtains for a given program M and number k all k-location counter-factual type errors. The function *get_cft_nloc* uses the function `LEAFLOCS` to determine all (leaf) locations of the program, the function `SUBSETS` to determine all subsets of k elements, and applies the *get_cft* function with the standard function `MAP` to every subset (represented as list).

The function *get_cft* obtains for a term M and list of locations L a counter-factual type error, namely an actual and expected type for each location. However, it returns the empty list if no such counter-factual type error exists for the given arguments. Therefore function *get_cft_nloc* usually returns many empty lists. The function *get_cft* uses the functions *pierce* and *infer*.

The function *pierce* takes an expression and list of locations. It inserts new variables as holes at the given locations in the expression and also returns the list of new variables. The function `GENVAR` creates a fresh variable from a given location.

The function *infer* takes an expression with k "holes" and the list of "hole" variables; it returns a k-location counter-factual type error, if for the given input there is such a type error. Otherwise, it returns the empty list. The function `INFER` is an existing type checker. It takes an expression and returns its inferred type. However, if the expression is ill-typed, the type checker raises an exception `TYPE_ERROR`. In the definition of the function *infer* that exception is caught by the `try ... with` construct. Finally, the function `GET_LOC` obtains from a variable the location information given at its creation. Note that if type checking succeeds, then every "hole" variable will have a type of the shape *actual type* $\rightarrow$ *expected type*.

4 OBTAINING COUNTER-FACTUAL TYPES FOR THE LET-POLYMORPHIC λ-CALCULUS

Problem. The method presented in the preceding two sections does not work with let polymorphism. Consider the following ill-typed example:

$$
\begin{array}{rcll}
(M : \mathit{term}) & ::= & c^l & \text{(constant)} \\
 & | & x^l & \text{(variable)} \\
 & | & \mathtt{fun}\ x \mapsto M & \text{(abstraction)} \\
 & | & \mathtt{@}\, M_1\, M_2 & \text{(application)} \\
(l : \mathit{loc}) & ::= & \mathit{location\ information} & \\
(\tau : \mathit{typ}) & ::= & b & \text{(type variable)} \\
 & | & \mathrm{int}, \mathrm{bool}, \ldots & \text{(type constants)} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{(function type)}
\end{array}
$$

Figure 1: The terms and types of $\lambda^{\rightarrow}$

$$
\begin{array}{l}
\mathit{pierce} : \mathit{term} * (\mathit{loc}\ \mathit{list}) \rightarrow \mathit{term} * (\mathit{var}\ \mathit{list}) \\
\mathit{pierce}[\![c^l]\!]_L = \\
\quad \mathtt{if}\ l \in L\ \mathtt{then}\ \mathtt{let}\ u = \mathtt{GENVAR}\ l\ \mathtt{in}\ (\mathtt{@}\, u\, c^l, [u]) \\
\qquad \mathtt{else}\ (c^l, []) \\
\mathit{pierce}[\![x^l]\!]_L = \\
\quad \mathtt{if}\ l \in L\ \mathtt{then}\ \mathtt{let}\ u = \mathtt{GENVAR}\ l\ \mathtt{in}\ (\mathtt{@}\, u\, x^l, [u]) \\
\qquad \mathtt{else}\ (x^l, []) \\
\mathit{pierce}[\![\mathtt{fun}\ x \mapsto M]\!]_L = \\
\quad \mathtt{let}\ (M', us) = \mathit{pierce}[\![M]\!]_L\ \mathit{in}\ (\mathtt{fun}\ x \mapsto M', us) \\
\mathit{pierce}[\![\mathtt{@}\, M_1\, M_2]\!]_L = \\
\quad \mathtt{let}\ (M_1', us_1) = \mathit{pierce}[\![M_1]\!]_L\ \mathtt{in} \\
\quad \mathtt{let}\ (M_2', us_2) = \mathit{pierce}[\![M_2]\!]_L\ \mathtt{in}\ (\mathtt{@}\, M_1'\, M_2', us_1 + us_2)
\end{array}
$$

$$
\begin{array}{l}
\mathit{infer} : \mathit{term} * (\mathit{var}\ \mathit{list}) \rightarrow (\mathit{loc} * \mathit{typ} * \mathit{typ})\ \mathit{list} \\
\mathit{infer}[\![M]\!]_{[u_1;\cdots;u_n]} = \\
\quad \mathtt{try}(\mathtt{let}\ ((\tau_1 \rightarrow \tau_1') \cdots \rightarrow (\tau_n \rightarrow \tau_n') \rightarrow \tau) = \\
\qquad\quad \mathtt{INFER}\ (\mathtt{fun}\ u_1 \mapsto \cdots \mapsto \mathtt{fun}\ u_n \mapsto M)\ \mathtt{in} \\
\qquad [(\mathtt{GET\_LOC}\ u_1, \tau_1, \tau_1'); \cdots ; (\mathtt{GET\_LOC}\ u_n, \tau_n, \tau_n')]) \\
\quad \mathtt{with}\ \mathtt{TYPE\_ERROR} \mapsto [] \\
\\
\mathit{get\_cft} : \mathit{term} * (\mathit{loc}\ \mathit{list}) \rightarrow (\mathit{loc} * \mathit{typ} * \mathit{typ})\ \mathit{list} \\
\mathit{get\_cft}[\![M]\!]_L = \\
\quad \mathtt{let}\ (M', \mathit{locs}) = \mathit{pierce}[\![M]\!]_L\ \mathtt{in}\ \mathit{infer}\ [\![M']\!]_{\mathit{locs}} \\
\\
\mathit{get\_cft\_nloc} : \mathit{term} * \mathit{int} \rightarrow ((\mathit{loc} * \mathit{typ} * \mathit{typ})\ \mathit{list})\ \mathit{list} \\
\mathit{get\_cft\_nloc}[\![M]\!]_k = \\
\quad \mathtt{MAP}\ (\lambda L.\ \mathit{get\_cft}[\![M]\!]_L)\ (\mathtt{SUBSETS}\ (\mathtt{LEAFLOCS}\ M)\ k)
\end{array}
$$

Figure 2: Obtaining counter-factual types for $\lambda^{\rightarrow}$

```
let id = (fun lst -> List.iter
                       (fun x -> x) lst) in
  (id [1;3;4], id [true])
```

Because the type of `List.iter` is `('a -> unit) -> 'a list -> unit`, the type of `id` must be `unit list -> unit`. However, we pass two non `unit` lists (`[1;3;4] : int list` and `[true] : bool list`) to `id`, so type checking fails. We assume that the use of `List.iter` is the source of the type error: the programmer intended to use the function `List.map` instead, which has type `('a -> 'b) -> 'a list -> 'b list`, and thus makes the whole program well-typed.

Let us now apply our old method to the program (changed parts are underlined):

```
(fun hole ->
let id = (fun lst -> (hole List.iter)
                      (fun x -> x) lst) in
  (id [1;3;4], id [true]))
```

We expect to obtain counter-factual types from this transformed program. However, it is ill-typed! In the original program `List.iter` has a polymorphic type, but the expression substituted for it, `(hole List.iter)`, has a monomorphic type, because the variable `hole` is λ-bound in the program. Therefore the let-bound variable `id` is monomorphic too and its two different uses have to cause a type error.

Solution. To preserve the polymorphism of let-bound variable `id`, we move the binding for `hole` under the definition of `id`.

```
let id = (fun hole -> (fun lst ->
           (hole List.iter) (fun x -> x) lst))
```

This transformation increases the number of `id`'s arguments. Therefore, we add additional variables `hole1` and `hole2` to each occurrence of `id` as the first argument.

```
let id = (fun hole -> (fun lst ->
          (hole List.iter) (fun x -> x) lst)) in
  (id hole1 [1;3;4], id hole2 [true])
```

In this program the fresh variables `hole1` and `hole2` are not bound. Therefore we add lambda bindings for them and obtain the following program as the final result:

```
fun hole1 -> fun hole2 ->
let id = (fun hole -> (fun lst ->
          (hole List.iter) (fun x -> x) lst)) in
  (id hole1 [1;3;4], id hole2 [true])
```

We infer the type of this transformed program using an existing type checker and obtain the expected types and actual types of the occurrence of `List.iter`. From the types of `hole1` and `hole2` we know that one of the expected types is `('b -> 'b) -> int list -> 'c` and another one is `('d -> 'd) -> bool list -> 'e`. Using these types, we can produce the following counter-factual type error message:

```
let id = (fun lst -> List.iter (fun x -> x) lst) in
  (id [1;3;4], id [true])

Error: This expression has type
    (('a -> unit) -> 'a list -> unit)
but it should be an expression with types
    ('b -> 'b) -> int list -> 'c
    ('d -> 'd) -> bool list -> 'e
```

We note that a counter-factual type error sometimes has to explicitly require that the replacement for an expression is polymorphic. Our message states this requirement by listing several types for the replacement.

Fixed Points. In OCaml the let-rec-binding is also polymorphic. However, we can translate an OCaml program such as

```
let rec exponential x =
  if x = "0" then 1
             else exponential (x - 1) in
   exponential 5
```

into our language as follows:

$$
\begin{array}{rcll}
(M : term) & ::= & c^l & \text{(constant)} \\
 & | & x^l & \text{(variable)} \\
 & | & \texttt{fun}\ x \mapsto M & \text{(abstraction)} \\
 & | & \texttt{@}\, M_1\, M_2 & \text{(application)} \\
 & | & \texttt{let}\ x = M_1\ \texttt{in}\ M_2 & \text{(let expression)} \\
 & | & \texttt{fix M} & \text{(fixed point)}
\end{array}
$$

Figure 3: The terms of λ^{let}

$$
\begin{array}{l}
\mathit{pierce} : \mathit{term} * (\mathit{loc}\ \mathit{list}) \to \mathit{term} * \mathit{var}\ \mathit{list} \\
\mathit{pierce}[\![\texttt{let}\ x = M_1\ \texttt{in}\ M_2]\!]_L = \\
\quad \texttt{let}\ (M_1', [u_1; \cdots; u_n]) = \mathit{pierce}[\![M_1]\!]_L\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us') = \mathit{add}[\![M_2]\!]_{(x,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2'', us'') = \mathit{pierce}[\![M_2']\!]_L\ \texttt{in} \\
\quad (\texttt{let}\ x = \texttt{fun}\ u_1 \mapsto \cdots \mapsto \texttt{fun}\ u_n \mapsto M_1'\ \texttt{in}\ M_2'', us' + us'') \\
\mathit{pierce}[\![\texttt{fix}\, M]\!]_L = \\
\quad \texttt{let}\ (M', us) = \mathit{pierce}[\![M]\!]_L\ \texttt{in}\ (\texttt{fix}\, M', us) \\
\\
\mathit{add} : \mathit{term} * \mathit{var} * \mathit{int} \to \mathit{term} * \mathit{var}\ \mathit{list} \\
\mathit{add}[\![c^l]\!]_{(y,n)} = (c^l, []) \\
\mathit{add}[\![x^l]\!]_{(y,n)} = \\
\quad \texttt{if}\ \ y \neq x\ \texttt{then}\ (x^l, []) \\
\qquad \texttt{else let}\ u_1 = \texttt{GENVAR}\ l\ \texttt{in} \\
\qquad\qquad \vdots \qquad \vdots \\
\qquad\qquad \texttt{let}\ u_n = \texttt{GENVAR}\ l\ \texttt{in} \\
\qquad\qquad (\texttt{@}\, x\, u_1\ \ldots\ u_n, [u_1; \cdots; u_n]) \\
\mathit{add}[\![\texttt{fun}\ x \mapsto M]\!]_{(y,n)} = \\
\quad \texttt{if}\ y = x\ \texttt{then}\ (\texttt{fun}\ x \mapsto M, []) \\
\qquad \texttt{else let}\ (M', us') = \mathit{add}[\![M]\!]_{(y,n)}\ \texttt{in} \\
\qquad\quad (\texttt{fun}\ x \mapsto M', us') \\
\mathit{add}[\![\texttt{@}\, M_1\, M_2]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M_1', us_1) = \mathit{add}[\![M_1]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us_2) = \mathit{add}[\![M_2]\!]_{(y,n)}\ \texttt{in}\ (\texttt{@}\, M_1'\, M_2', us_1 + us_2) \\
\mathit{add}[\![\texttt{let}\ x = M_1\ \texttt{in}\ M_2]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M_1', us_1) = \mathit{add}[\![M_1]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us_2) = \mathit{add}[\![M_2]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{if}\ us_1 = []\ \texttt{then}\ (\texttt{let}\ x = M_1'\ \texttt{in}\ M_2', us_2) \\
\qquad \texttt{else let}\ [u_1; \cdots; u_m] = us_1\ \texttt{in} \\
\qquad\quad \texttt{let}\ (M_2'', us_2') = \mathit{add}[\![M_2']\!]_{(x,m)}\ \texttt{in} \\
\qquad\quad (\texttt{let}\ x = \texttt{fun}\ u_1 \mapsto \cdots \texttt{fun}\ u_m \mapsto M_1'\ \texttt{in}\ M_2'' \\
\qquad\quad , us_2 + us_2') \\
\mathit{add}[\![\texttt{fix}\, M]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M', us') = \mathit{add}[\![M]\!]_{(y,n)}\ \texttt{in}\ (\texttt{fix}\, M', us')
\end{array}
$$

Figure 4: Obtaining counter-factual types for λ^{let} (new cases only)

```
let exponential =
  (fix (fun f -> fun x ->
    if x = "0" then 1 else f (x - 1))) in
  exponential 5
```

The fixed point construct is monomorphic. `fix` simply has type `(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)` and does not

require any special treatment. Only the let-binding is polymorphic and needs to be handled as described before. For example, when we consider the occurrence of `"0"` in the program as a potential type error location, we pass the following transformed program to the type checker:

```
(fun hole1 ->
let exponential =
  (fun hole -> (fix (fun f -> fun x ->
    if x = (hole "0") then 1 else f  (x - 1)))) in
  exponential hole1 5 )
```

From the inferred type for `hole1` we then produce the counter-factual type error message:

```
let exponential =
  (fix (fun f  -> fun x ->
    if x = "0" then 1 else f (x - 1))) in
  exponential 5

Error: This expression has type
    string
but it should be an expression of type
    int
```

Formalisation. Figure 3 shows the syntax of the let-polymorphic λ-calculus λ^{let}. The types are the same as for $\lambda^{\rightarrow}$ in Figure 1. We do not need to include type schemas amongst the types. Type schemas are used during type inference for inferring polymorphic types. Because we use an existing compiler's type checker and its inferred types, our type debugger never sees any type schemas.

Figure 4 extends our algorithm of Figure 2 to obtaining counter-factual types for λ^{let}. Recall the purpose of function *pierce*: It takes an expression and list of locations; then it inserts new variables as holes at the given locations in the expression and also returns the list of new variables. For a let-expression it uses the new function *add* to add new n argument variables to every occurrence of a target variable y. The function *add* returns both the transformed expression and the list of new argument variables. Note that `GENVAR` is not a pure function but creates a fresh variable for every call.

5 OBTAINING COUNTER-FACTUAL TYPES MORE EFFICIENTLY

In Sections 3 and 4 we introduced a method to obtain counter-factual types using an existing type checker. However, the time complexity of the naive implementation of this method (*get_cft_nloc* in Figure 2) is too high to be used for anything but short toy programs. Hence we consider in this section how to improve the algorithm to make the method applicable to real programs.

Time complexity of the naive implementation. Assume we have a program of size n. Any leaf in the syntax tree could be a single location type error. So to find all single location errors, we have to call the type checker $O(n)$ times with a variant of the program. There are $O(n^2)$ location pairs that could be 2-location type errors. In general, the naive implementation described in Sections 3 and 4 calls the type checker $O(n^k)$ times to find all k-location counter-factual type errors. This time complexity is clearly unacceptable in practice.

Type constraints and program slices. To improve the time complexity, we now build on established work on type error slicing, which is also discussed in Section 8. A slice is a part of a program, possibly including holes. Every part of a program gives rise to a constraint on the types of the whole program. For a fixed program there is a bijection between its slices and the subsets of its type constraints. We assume that we have an ill-typed program P and its set of type constraints is p. Because P is an ill-typed program, $ill\_typed(p)$ holds.

If we assume that the original type constraints of the holed part (for counter-factual types) are p_{cft}, then the type constraints of the transformed program are $p \setminus p_{\text{cft}}$. The aim of this paper is to find p_{cft} that satisfies the following property:

$$well\_typed(p \setminus p_{\text{cft}}) \; and \; \forall p' \subset p_{\text{cft}}.\; ill\_typed(p \setminus p')$$

The second part ensures that p_{cft} is minimal; a minimal set is removed from the program to make the program well-typed.

Our concern is that it is sometimes hard to find p_{cft}. The key observation to solve this problem is that this property is similar to the property of type error slices [6]. Let p_{slice} be the type constraints of a type error slice of P. Then p_{slice} satisfies the following property:

$$ill\_typed(p_{\text{slice}}) \; and \; \forall p' \subset p_{\text{slice}}.\; well\_typed(p')$$

From this definition we know that if we have a type error slice p_{slice}, we can obtain many p_{cft}s of p_{slice} easily. Therefore, our improved approach is the following. First we obtain a type error slice p_{slice}. Any non-empty subset of p_{slice} is p_{cft} of p_{slice}. To obtain p_{cft} of p, first we choose a part of p_{slice} as p_{cft} and restore each element of $p \setminus p_{\text{slice}}$ to p_{slice} or p_{cft}. Then we can obtain p_{cft} of p. If we apply this method to each subset of p_{slice}, then we can obtain several counter-factual types of the program P.

An example. Let us see how our improved approach works with a concrete example:

```
(fun x -> x +. 1) 3
```

The underlined part is a type error slice of the program. Every part of a type error slice can be a hole for a counter-factual type error. Thus we can choose either `+.` or `1` as a `hole`.

Choice 1. First we pierce 1, that is, apply the *pierce* function to the program with the location of 1.

```
(fun hole -> (fun x -> x +. (hole 1)) 3)
```

We might expect this program to be well-typed, but it is not, because the types of `3` and `+.` conflict. This is because if $well\_typed(p_{\text{slice}} \setminus x)$ holds, $well\_typed(p \setminus x)$ does not always hold. To avoid this problem, we should pierce the parts that are not included in the type error slice by `holes` as follows:

```
(fun hole -> (fun h1 -> (fun h2 ->
  (fun x -> h1 +. (hole 1)) h2)))
```

Then we obtain a counter-factual type error and generate the following error message:

```
(fun x -> x +. 1) 3
Error: This expression has type
    int
but it should be an expression of type
    float
```

This approach, making counter-factual types of type error slices, seems to work well. However, this is a counter-factual type of this type error slice, it is not always a counter-factual type of the whole program. To see the problem, let us consider another choice of the original program.

Choice (`+.`). Following our method, we obtain:

```
(fun hole -> (fun h1 -> (fun h2 ->
  (fun x -> (hole (+.)) h1 1) h2)))
```

The parts that are not included in the type error slice are replaced by holes too. Because this program is well-typed, we can produce the following message:

```
(fun x -> x +. 1) 3
Error: This expression has type
    float -> float -> float
but it should be an expression of type
    'a -> int -> 'b
```

Because this is a counter-factual type error for this type error slice, the expected type is more general than the expected type of the whole program, `int -> int -> 'c`. However, it is very easy to expand the upper program to obtain the counter-factual types of the whole program. We just restore removed program fragments that do not conflict. In this example, we can restore all removed program fragments, `x` and `3`.

```
(fun hole -> (fun x -> (hole (+.)) x 1) 2)
```

This program is well-typed and satisfies the property of counter-factual type errors of the whole program.

The algorithm. We need a type error slicer using the existing compiler's type checker. Schilling [12] computes type error slices by increasing program fragments until the program is ill-typed. Because the last added program fragment relates to a type error, we can obtain type error slices by collecting them. We can emulate his approach using the function *pierce* of Figures 2 and 4. Figure 5 shows the functions for type error slicing, Figure 6 shows the function for restoring, and the main function for improved obtaining counter-factual types is given in Figure 7.

The slicing function increases the number of program fragments and checks whether the program fragments are sufficient to be ill-typed. The function $slicing'$ chooses a location randomly from L using `PICKUP_ONE`. Because L is the locations of holes, $L_{\text{slice}} \setminus l$ means increasing l's program fragment (= decreasing the number of holes). If the program constructed by program `M` and location information $L_{\text{slice}} \setminus l$ is ill-typed, then the last added location relates to the type error. Therefore $slicing'$ returns the last added location. The function *slicing* gathers the return values of $slicing'$. It stops when the gathered locations are sufficient to be ill-typed.

The function *restore* restores program parts that were removed by type error slicing. It receives locations of counter-factual types for a type error slice and checks whether each location makes the whole program well-typed or not. If adding a location preserves well-typedness, then the location is not related to a type error.

The function *get_cfts_of_slice* is the main function; it obtains counter-factual types. First, it obtains a type error slice using *slicing*. Because L_{slice} includes the locations of a type error slice, we can use $L \setminus L\_\text{slice}$ for obtaining a type error slice using *pierce*. We add a part of L_{slice} to it for obtaining its counter-factual type, and restore the L_{cft} and infer its type. We apply this method to each L_{cft} and obtain several counter-factual types.

$$
\begin{array}{l}
slicing' : term * (loc\ list) * (loc\ list) \to loc \\
slicing'[\![M]\!]_{(L, L_{\text{slice}})} = \\
\quad \texttt{let}\ l = \texttt{PICKUP\_ONE}\ L\ \texttt{in} \\
\quad \texttt{if}\ infer(M, L_{\text{slice}} \setminus l) = [] \\
\qquad \texttt{then}\ slicing'[\![M]\!]_{(L \setminus l, L_{\text{slice}} \setminus l)} \\
\qquad \texttt{else}\ l
\end{array}
$$

$$
\begin{array}{l}
slicing : term * (loc\ list) * (loc\ list) \to loc\ list \\
slicing[\![M]\!]_{(L, L_{\text{slice}})} = \\
\quad \texttt{if}\ infer(M, L_{\text{slice}}) = []\ \texttt{then}\ L_{\text{slice}} \\
\qquad \texttt{else let}\ l = slicing'[\![M]\!]_{(L, L_{\text{slice}})}\ \texttt{in} \\
\qquad\quad slicing[\![M]\!]_{(L \setminus l, L_{\text{slice}} \setminus l)}
\end{array}
$$

Figure 5: Type error slicing

$$
\begin{array}{l}
restore : term * (loc\ list) * (loc\ list) \to loc\ list \\
restore[\![M]\!]_{(L_{\text{cft}}, L_{\text{acc}})} = \\
\quad \texttt{if}\ L_{\text{cft}} = []\ \texttt{then}\ L_{\text{acc}} \\
\qquad \texttt{else let}\ l = \texttt{PICKUP\_ONE}\ (L_{\text{cft}})\ \texttt{in} \\
\qquad\quad \texttt{if}\ infer(M, L_{\text{cft}} \setminus l) = [] \\
\qquad\qquad \texttt{then}\ restore[\![M]\!]_{(L_{\text{cft}} \setminus l, L_{\text{acc}})} \\
\qquad\qquad \texttt{else}\ restore[\![M]\!]_{(L_{\text{cft}} \setminus l, L_{\text{acc}} + l)}
\end{array}
$$

Figure 6: Restoring

$$
\begin{array}{l}
get\_cfts\_of\_slice : term * (loc\ list) \to ((loc * typ * typ)\ list)\ list \\
get\_cfts\_of\_slice[\![M]\!]_L = \\
\quad \texttt{let}\ L_{\text{slice}} = slicing[\![M]\!]_{(L, L)}\ \texttt{in} \\
\quad \texttt{let}\ L_{\text{cft}}s = \texttt{MAP}(\lambda l.\ (L \setminus L_{\text{slice}}) + l)\ L_{\text{slice}}\ \texttt{in} \\
\quad \texttt{let}\ M's = \texttt{MAP}\ (\lambda L_{\text{cft}}.\ restore[\![M]\!]_{(L_{\text{cft}}, [])})\ L_{\text{cft}}s\ \texttt{in} \\
\quad \texttt{MAP}\ (\lambda M'.\ infer[\![(M', [l])]\!])\ M's
\end{array}
$$

Figure 7: Improved obtaining counter-factual types

Computational complexity. Let n be the program size. The complexity of type error slicing is $O(n^2)$ and the complexity of restoring is $O(n)$ for each p_{cft}. Therefore the whole complexity is $O(n^2)$. If we need k locations for $k > 1$, then the complexity is lower than the naive use of Sections 3 and 4.

6 AN INTERACTIVE TYPE DEBUGGER

In this section, we describe an interactive debugger that uses the enumerated counter-factual type error messages to find the correct type error location. Usually the debugger shows only a few counter-factual type error messages to the programmer.

6.1 When is the correct location found?

The purpose of the interactive type debugger is to determine a subexpression in the program that needs to be changed to obtain a well-typed program. The debugging process is driven by the

programmer's intention, the types that the programmer intends certain subexpressions to have.

To clarify the idea, let us consider some examples.

```
Choose your intended type for this expression.

 (1 +. 2)

A: float -> float -> float
B: int -> int -> int
Your choice: (C: another type): B
```

In this case `+.` is the correct type error location, because the actual type of `+.` and the programmer's intended type are in conflict.

```
Choose your intended type for this expression.

let f = (+.) in (f 1 2)

A: float -> float -> float
B: int -> int -> int
Your choice (C: another type): B
```

This program is very similar to the previous example. However, the cause of the type error has not yet been located. The highlighted expression `f` is not the cause, but the expression `(+.)`. Replacing `(+.)` by `(+)` makes the program well-typed.

These examples indicate when the cause of a type error has been found. During the debugging process the type debugger collects the programmer's intentions. The cause of a type error is located when there is a conflict amongst the stated programmer's intentions.

6.2 The debugging process.

We reconsider the following program from the introduction:

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

In this program we have four potential single counter-factual type error locations: `f` (in `f 2.0`), `2.0`, `^` and `n` (in `x ^ n`). Our interactive type debugger does not simply enumerate these four locations with actual and expected types and leaves it to the programmer to find the location that causes the type error. Instead the type debugger starts by randomly selecting one counter-factual type error message. Let us assume that the selected message is

```
Choose your intended type for this expression.

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

A: float
B: string
Your choice (C: another type):
```

Choice A. The programmer states that the actual type, `float`, is the intended type. The debugger adds this information to the original ill-typed program by adding a type annotation:

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

For this annotated program we can obtain only one counter-factual type error that does not include `(n:float)`. Because `^` and `(n:float)` have a type conflict, we have to remove either `^` or `(n:float)` for the program to be well-typed. However, the debugger already asked about `n`, hence `^` must be the type error location:

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

This expression has type
    string -> string -> string
but it should be an expression of type
    'a -> float -> 'b
```

In conclusion, we started with four potential type error locations, but after one question the debugger has identified the correct type error location.

Choice B. The programmer states that the expected type, `string`, is the intended type.

Note that just because the expected type is the intended type, `n` is not necessarily the cause of the type error. The types of variables are often forced by other parts of the program[3].

To judge whether the `n` is the cause of the type error or not, the type debugger generates the following program with several holes and passes it to the type checker:

```
(fun h5 -> fun h6 -> fun h7 -> fun h8 -> fun h9 ->
  fun h10 ->
    let f = fun h1 -> fun h2 -> fun h3 -> fun h4 ->
       fun n -> fun lst ->
         h1 (fun x -> h2 h3 (n:string)) h4 in
         h5 h6 h7 h8 h9 h10)
```

This is the original program with every leaf replaced by a different variable, except for the sub term `n` and its type annotation. Because this program is well-typed, the debugger knows that it has to continue considering other candidates as cause of the type error.

The debugger adds the intended type as a type annotation and produces counter-factual type error messages for the program:

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

There are only two single-location counter-factual type error messages, one for `f` and one for `2.0`. Note that `^` is no longer a candidate.

The debugger randomly chooses one of the two counter-factual type error messages:

```
Choose your intended type for this expression.

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

[3] This does not mean that the cause of a type error has to be in another part. The cause is sometimes a variable itself. In this example, if the programmer replaced `n` by another variable that has type `string`, then the whole program would be well-typed.

```
A: float
B: string
Your choice (C: another type): B
```

Let us assume that the programmer states that the expected type B is correct. Integrating this intention into the program the debugger internally produces

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

Again the debugger has to check whether the cause has been found or it needs to continue asking questions. Hence again it replaces every leaf by a different variable excluding the terms already asked about and their type annotations. It passes the following program to the type checker:

```
(fun h5 -> fun h6 -> fun h7 -> fun h8 -> fun h9 ->
let f = fun h1 -> fun h2 -> fun h3 -> fun h4 ->
  fun n -> fun lst ->
  h1 (fun x -> h2 h3 (n:string)) h4 in
  h5 h6 h7 h8 h9 (2.0:string))
```

This program is ill-typed and hence the last counter-factual type error did indeed locate the fault:

```
Type error located:

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

This expression has type
    float
but it should be an expression of type
    string
```

Type annotations. Type annotations reduce type error candidates very effectively. This raises the question whether, to find the cause of a type error, the programmer could not simply add themselves type annotations about their intentions to a program. Sometimes this does work, but there is no guarantee. Type checking algorithms are free to handle type annotations in many different ways. When the OCaml type checker traverses a type annotation, it often already inferred the type of the annotated term. Hence OCaml often identifies the correctly annotated term as the cause of the type error. Because counter-factual types are produced independently of the order of type unification, type annotations work more effectively than with standard type checking.

6.3 The Type Error Debugging Algorithm

We extend the object language with type annotated terms in Figure 8. For simplicity we assume that annotated terms are only introduced by the debugger[4].

[4]The programmer's annotated terms might be the cause of a type error, but the answers to the debugger cannot be the cause of a type error, provided the debugger's questions are answered correctly. Thus, if we want to introduce the programmer's type annotated terms, we should distinguish between these two forms of annotations.

$$(M : term) \quad ::= \quad (M : \tau) \qquad \text{(type annotated term)}$$

Figure 8: Additional syntax for type annotation

$$\begin{array}{l}
pierce : term * (loc\ list) \to term * (loc\ list) \\
pierce[\![(M : \tau)]\!]_L = \texttt{let}\ (M', us) = pierce[\![M]\!]_L\ \texttt{in}\ (M' : \tau, us) \\
\\
add : term * term * (loc\ list) \to term * (loc\ list) \\
add[\![(M : \tau)]\!]_{(f,us)} = \\
\quad \texttt{let}\ (M', us') = add[\![M]\!]_{(f,us)}\ \texttt{in}\ ((M' : \tau), us')
\end{array}$$

Figure 9: Additional definitions for obtaining counter-factual type errors

$$\begin{array}{l}
annot : term * loc * typ \to term \\
annot[\![c^l]\!]_{(l',\tau)} = \texttt{if}\ l = l'\ \texttt{then}\ (\mathsf{c}^l : \tau)\ \texttt{else}\ \mathsf{c}^l \\
annot[\![x^l]\!]_{(l',\tau)} = \texttt{if}\ l = l'\ \texttt{then}\ (x^l : \tau)\ \texttt{else}\ x^l \\
annot[\![\texttt{fun}\ x \mapsto M]\!]_{(l',\tau)} = \texttt{fun}\ x \mapsto annot[\![M]\!]_{(l',\tau)} \\
annot[\![\texttt{@}\ M_1\ M_2]\!]_{(l',\tau)} = \texttt{@}\ annot[\![M_1]\!]_{(l',\tau)}\ annot[\![M_2]\!]_{(l',\tau)} \\
annot[\![\texttt{let x = M}_1\ \texttt{in M}_2]\!]_{(l',\tau)} = \\
\quad \texttt{let}\ x = annot[\![M_1]\!]_{(l',\tau)}\ \texttt{in}\ annot[\![M_2]\!]_{(l',\tau)} \\
annot[\![\texttt{fix}\ M]\!]_{(l',\tau)} = \texttt{fix}\ annot[\![M]\!]_{(l',\tau)} \\
\\
debug : term * ((loc * typ)\ list) * (loc\ list) \to unit \\
debug[\![M]\!]_{(asked, remaining)} = \\
\quad \texttt{if}\ infer(pierce[\![M]\!]_{remaining}) = [] \\
\qquad \texttt{then ()} \quad (*\ \text{located at last question}\ *) \\
\qquad \texttt{else let}\ lst = get_cft_of_slice[\![M]\!]_{remaining}\ \texttt{in} \\
\qquad\quad \texttt{let}\ (l, \tau, \tau') = \texttt{PICKUP_ONE}\ lst\ \texttt{in} \\
\qquad\quad \texttt{let}\ \tau_{\text{intended}} = \texttt{ASK}[\![M]\!]_{(l,\tau,\tau')}\ \texttt{in} \\
\qquad\quad \texttt{let}\ M' = annot[\![M]\!]_{(l,\tau_{\text{intended}})}\ \texttt{in} \\
\qquad\quad debug[\![M']\!]_{(l,\tau_{\text{intended}})::asked, remaining \setminus l)}
\end{array}$$

Figure 10: Interactive type debugger using type annotations

The definitions for additional syntax to obtain the counter-factual types are shown in Figure 9. In the definition of *pierce* we do never remove type annotations, because we assume type annotations introduced by the debugger are always correct.

The definitions for a type debugger using counter-factual types and type annotations are shown in Figure 10. The function *annot* adds the user's intentions as type annotations. It receives a program, the target location l', and the user's intended type τ for l'. For constants and variables, it adds a type annotation if its location is equal to the target location. For other constructors we call *annot* recursively to add a type annotation in a sub-expressions.

The function *debug* is an interactive type debugger. It receives a closed ill-typed program M, a list of already asked locations with their intended types *asked* and not asked locations *remaining*. First, we check whether the already asked parts of the annotated program M cause a type conflict or not. If they cause a type conflict, then the

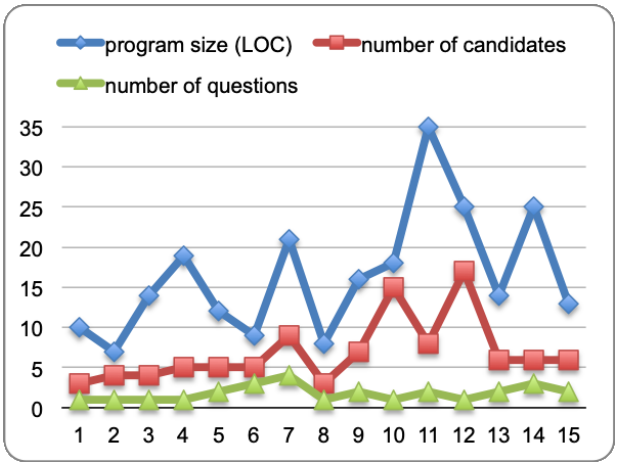

Figure 11: Search space of our prototype

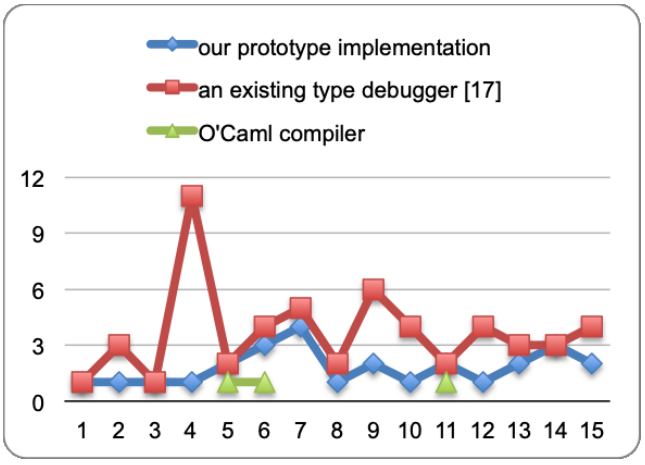

Figure 12: Question numbers of different type debuggers

type debugging process ends. Otherwise, we need to continue asking questions. For asking questions, we obtain counter-factual types using *get_cft_of_slice* and ask the programmer about it. The external function `ASK` receives the program M and its counter-factual type, asks the programmer and returns the programmer's intended type τ_{intended}. After that, the type debugger adds the correct intended annotation to the program and obtains a new annotated program `M'`. Because the cause of the type error is not located yet, we call *debug* again with the updated information. To start type debugging, we call *debug* with the original ill-typed program M, an empty asked list and a list of all locations of M.

7 EVALUATION

We built a prototype implementation of our type debugging method. It is embedded in OCaml 4.01.0. We compared it with two other implementations: OCaml 4.01.0 itself and an existing interactive type debugger [17], which is also embedded in OCaml 4.01.0. We use 15 small ill-typed programs from two sources: student's programs from an OCaml introductory course and test code for Skalpel's online demonstration [11].

How was the search space reduced by our approach? Figure 11 gives for each program its size, the number of candidates for the first question and the number of questions needed for locating the fault. We can learn two things: Program parts that are unrelated to type errors are often successfully removed. If a type error includes many conflicts, then the number of candidates can be large, but the numbers of questions needed is smaller, often substantially so.

For removing the unrelated candidates, the user's interactive input can help much. In Test 11 the number of candidates is 8 and our prototype asks about the type of `::`. With the general type of `::`, the number of candidates for the second question is 7. However, if the user inputs a specific type (e.g., `int → int list → int list`), then the number of the candidates for the second question becomes 1. Thus we know that the type annotations of Figure 10 remove candidates substantially.

Program size (LOC)	Error line (LOC)	Time (seconds)
10	35	1.01
22	37	2.69
122	5	2.43
122	39	2.85
122	107	3.66
482	413	6.92

Table 1: Runtime measurements

Is the prototype better than an existing type debugger? In Figure 12 we compare the number of questions with an existing interactive type debugger [17] and the OCaml compiler. Because the OCaml compiler is not interactive, we only record when its error message locates the source of the type error correctly (Test 5, 6 and 11). We see that our prototype locates the source of a type error quicker than the older debugger. This is because the older debugger asks about many parts of the program, but our type debugger focusses on parts related to the type error. The OCaml compiler can sometimes locate the error source, but not always.

Is runtime reasonable for interactive use? Table 1 shows the runtimes of our prototype debugger for different programs. These programs were made by injecting a type error into a well-typed program. The runtime time depends on the program size and error location in the program; it varies substantially. Because our implementation is a prototype, we expect to achieve speed improvements in the future. At least our prototype works in reasonable time for small programs (around 100 LOC).

8 RELATED WORK

A wealth of papers have been published on type error debugging since the 1980's. Here we focus on a few.

New Type Checking Algorithms. Existing functional programming systems use variants of the standard type checking algorithm $\mathcal{W}$ by Milner and Damas [5]. Algorithm $\mathcal{W}$ traverses the abstract syntax tree of the program and eagerly solves type constraints by

unification. When type unification fails, the currently inspected expression is reported as type error location together with its actual and expected type. Thus the reported type error location depends on the order in which $\mathcal{W}$ solves type constraints, that is, traverses subexpressions and unifies types.

Many improved type checking algorithms have been proposed. Wand [19] extends $\mathcal{W}$ to keep track of the history of how type variables are instantiated and shows this history for conflicting types when unification fails. Lee and Yi [9] propose to use the algorithm $\mathcal{M}$. They show that $\mathcal{M}$ finds type conflicts earlier than algorithm $\mathcal{W}$ and thus $\mathcal{M}$ reports a smaller expression as error location. Yang et al. [20] describe a new algorithm $\mathcal{IEI}$ that combines the algorithm $\mathcal{M}$ with unification of type assumption environments. Their implementation enumerates several error messages with counter-factual types like our method. However, $\mathcal{IEI}$ focuses on a type conflict in function applications. Therefore their approach does not enumerate all counter-factual type errors of the ill-typed program. In contrast to all these algorithms, our approach is completely independent of the order of traversing the syntax tree or solving type constraints.

Reusing an Existing Type Checker. Braßel [1] was the first to propose using an existing type checker for type debugging. His experimental tool TypeHope automatically corrects type errors for the functional logic language Curry. The type debugger Seminal [10] takes this idea further. It replaces erroneous parts with various syntactically correct similar expressions, and sees if they type check. If they do, the replacements are displayed as candidates for fixing the type error.

Chen and Erwig [2] already note that usually there exist too many different expression changes that correct a type error; most of these expression changes do not agree with the programmer's intentions. Hence, unlike Seminal [10] but like Chen and Erwig [2], our aim is to suggest type changes and leave it to the programmer to select the appropriate expressions.

Sharrad et. al. [13] take the idea of using an existing type checker as a black box even further by not transforming a syntax tree but just the lines of the original program text. They apply the delta-debugging method to locate a line or lines of code that cause the error. Scaling up to large programs is under current development.

Counter-Factual Types. Most existing compilers for Hindley-Milner-based programming languages produce type error messages with counter-factual types. Also several of the early proposals for improving type error debugging use counter-factual types. However, Chen and Erwig [2] were the first to clearly identify the concept of counter-factual types and argue their usefulness for type error debugging. They propose to assist type debugging by automatically enumerating all potential type error locations with counter-factual types. So a message suggests changing the actual type of a given program location to the counter-factual type. Chen and Erwig use a new type checking algorithm based on variational types. Because these variational types are monomorphic, they restrict counter-factual types to be monomorphic as well. Potential type error locations that require polymorphic types are not identified. In contrast, our approach produces in such a case a list of expected monomorphic types, which could also be transformed into a single expected polymorphic type.

Interactive Type Debugging. Interactive type debugging systems have been proposed to enable the programmer to include their intentions in the search for the error location. Chitil [4] developed an algorithmic debugger for type debugging, using a compositional type inference algorithm. Based on his work, Tsushima and Asai [17] designed an algorithmic type debugger for OCaml that uses the compiler's own type checker rather than a tailor-made type checking algorithm. Algorithmic debugging guarantees to find a type error location correctly, but answering the questions of an algorithmic debugger requires a good understanding of types, especially intended types, by the programmer.

Chen and Erwig [3] propose an interactive type debugger based on enumerating counter-factual type errors. Like our interactive type debugger it determines the correct type error location and gives its counter-factual type. However, the interaction is different in that the programmer does not choose between types like in our debugger but has to enter intended types for given subexpressions. Internally Chen and Erwig's type debugger relies on the use of their variational types [2] to quickly reduce the number of type error location candidates.

Type Error Slicing. The aim of type error slicing is to determine a minimal slice of an ill-typed program which contains all the program parts responsible for the ill-typedness. To make the program well-typed, a change within this type error slice is required. Underlying type error slicing is the observation that every program fragment corresponds to a constraint on the types of subexpressions of the program. For an ill-typed program the complete set of its type constrains is unsatisfiable. A minimal unsatisfiable set of type constraints can be computed that then corresponds to a minimal type error slice of the program.

Haack and Wells [6] define and implement type error slicing for ML using their own type checking algorithm which solves annotated type constraints. Later Rahli et al. [11] extend this work to a much larger fragment of ML. Independently Stuckey, Sulzmann, and Wazny [14][15][16] develop the idea of finding the source of type errors by solving constraints expressed with constraint handling rules (CHRs). Their type debugger called Chameleon implements its own type inference based on CHRs. The debugger can explain type errors and inferred types by showing relevant slices. Their work covers many advanced language features, such as type-annotation, Haskell-style overloading and generalised algebraic data types (GADTs).

Later Schilling [12] obtains type error slices for a large subset of Haskell using the compiler's type checker as a black box. Tsushima and Asai [18] improve Schilling's type error slicer using weights. If a type error slice has conflicts with several parts of the program, then it is more likely to be the cause of ill-typedness. Hence weighting such conflicts enables choosing better slices.

The advantage of type error slicing is that the process is fully automatic and the programmer does not have to answer any questions. On the other hand even minimal slices can still be relatively big and slices do not explain an error or how to correct it. Although looking very different to the programmer, type error slicing and our approach are closely related, as Section 5 demonstrates.

Heuristics for type debugging. Heuristics based on a large corpus of ill-typed programs can help with finding the correct type error

location and providing more helpful type error messages. Heeren and Hage [8] use a constraint-based type checker to flexibly vary the order of solving constraints based on a heuristic. Their approach sometimes but not always produces good error messages, depending on whether a specific type error is considered by the heuristic or not. Hage and Heeren [7] define heuristics for type debugging using their students' programs. They focus on helping non-expert programmers. Wherever our interactive type debugger currently makes a random choice it could instead be guided by a heuristic. With experience in practice the messages themselves could be made more helpful in future.

9 CONCLUSION AND FUTURE WORK

In this paper we proposed a new method for enumerating counter-factual type error messages for all potential type error locations in a program, reusing an existing type checker. Our method also fully supports the polymorphic Hindley-Milner type system in that it handles the situation where a subexpression of a program needs to be replaced by a polymorphic expression. The basic idea underlying our method is very simple: we introduce `holes` in the program and obtain their types. We introduced an improved method for obtaining counter-factual type error messages using type error slicing. Its computational complexity is $O(n^2)$ where n is the program's size. We married the enumeration of counter-factual type error messages with a new interactive debugger that locates the correct type error. Type annotations are at the heart of this interactive debugger.

We both gave informal descriptions with examples of all algorithms and defined them formally for the Hindley-Milner-typed λ-calculus with recursion. We also implemented it for a small subset of OCaml.

The advantage of this work is twofold. First, our method does not depend on any algorithm for inferring types and it requires only the inferred types. Hence our method can be applied to many programming languages based on the Hindley-Milner type system, regardless of which type inference algorithm is used by their compilers. Second, it is easy to implement. To implement a type checker that returns exactly the same type as the compiler's type checker would be tedious and error-prone.

We have several plans for future work:

We want to explore whether replacing the program slicing by our delta-debugging implementation [13] improves the speed of our type error debugger.

If an expression should be polymorphic, our counter-factual type error message currently lists several expected types. Anti-unification of these expected types might give the right expected polymorphic type.

We want to apply our idea to advanced language features. Practical functional programming languages include numerous features, for example module and object systems, for which we still have to define our method. Thus we expect to prove the scalability of our approach.

Finally, we want to develop heuristics for our type debugger. Which potential type error location should the debugger ask about first? Are some locations more likely to be correct than others? To get answers to these questions we need to implement and use a type debugger that covers a large subset of OCaml.

REFERENCES

[1] Braßel, B. "Typehope: There is hope for your type errors," *IFL 2004*.

[2] Chen, S., M. Erwig. "Counter-Factual Typing for Debugging Type Errors," *POPL 2014*, ACM.

[3] Chen, S., M. Erwig. "Guided Type Debugging," *FLOPS 2014*, pp. 35–51.

[4] Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," *ICFP 2001*, ACM, pp. 193–204.

[5] Damas, L., R. Milner. "Principal type-schemes for functional programs," *POPL 1982*, ACM, pp. 207–212.

[6] Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP'03)*, Volume 50 Issue 1-3 (2004).

[7] Hage, J., and B. Heeren. "Heuristics for Type Error Discovery and Recovery," *IFL 2007*, LNCS 4449, pp. 199-216.

[8] Heeren, B., D. Leijen., J. Hage. "Helium, for Learning Haskell," *Workshop on Haskell 2003*, ACM, pp. 62–72.

[9] Lee, O., K. Yi. "Proofs about a Folklore let-polymorphic Type Inference Algorithm," *ACM Transactions on Programming Languages and Systems*, pp. 707-723 (1998).

[10] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. "Searching for Type-Error Messages," *PLDI 2007*, pp. 425–434.

[11] Rahli, V., J. B. Wells, J. Pirie and F. Kamareddine. "Skalpel: a type error slicer for standard ML," *Electronic Notes in Theoretical Computer Science*, Vol. 312, pp. 197–213 (2015).

[12] Schilling, T. "Constraint Free Type Error Slicing," *TFP 2011*, pp. 1–16.

[13] Sharrad, J., O. Chitil and M.Wang. "Delta Debugging Type Errors with a Blackbox Compiler," *IFL 2018*, pp. 13–24.

[14] Stuckey, P. J., M. Sulzmann, J. Wazny. "Interactive type debugging in Haskell," *Workshop on Haskell 2003*, ACM, *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell'03)*, pp. 72–83.

[15] Stuckey, P. J., M. Sulzmann, J. Wazny. "Improving type error diagnosis," *Workshop on Haskell 2004*, ACM, pp. 80–91.

[16] Stuckey, P. J., M. Sulzmann, J. Wazny. "Type Processing by Constraint Reasoning," *APLAS 2006*, pp. 1–25.

[17] Tsushima, K., K. Asai. "An Embedded Type Debugger," *IFL 2012*, pp. 190–206.

[18] Tsushima, K., and K. Asai. "A weighted type-error slicer (in Japanese)," *Journal of Computer Software*, Vol. 31, No. 4, pp. 131–148 (2014).

[19] Wand, M. "Finding the Source of Type Errors," *POPL 1986*, ACM, pp. 38–43.

[20] Yang, J., G. Michaelson, P. Trinder, J. B. Wells. "Improved Type Error Reporting," *IFL 2000*, pp. 71-86.