Enumerating Counter-Factual Type Error Messages with an Existing Type Checker

Kanae Tsushima

Kyoto University
Research Fellow of
the Japan Society for the Promotion, PD
tsushima@fos.kuis.kyoto-u.ac.jp

Olaf Chitil

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

概 要 The type error message of a standard type checker for a functional language gives only a single location as potential cause of the type error. If that location is not the cause, which often is the case, then the type error message hardly helps in locating the real cause. Here we present a method that uses a standard type checker to enumerate locations that potentially cause the type error, each with an actual and a counter-factual type for the given location. Adding our method to existing compilers requires only limited effort but improves type error debugging substantially.

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 advantageous, 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 how.

First, we note that no system can fully automatically correct a type error, because the system cannot know the intentions of the programmer. Trivially, any ill-typed program could be replaced by some well-typed program, e.g. `x = 42`. Thus the type error disappears. Even restricting ourselves to smaller, "reasonable" changes of the program, the problem of unknown intentions persists. Consider the following OCaml program:

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

The program[1] is ill-typed. Because `^` is string concatenation in OCaml, the type of `f` is `string -> string list -> string list`. However, the program applies `f` to the floating point constant `2.0`.

At least two different reasonable locations for correcting the type error spring to mind. First, the floating point constant `2.0` could be wrong and should be replaced by some string. Only the programmer can know which string they want, maybe `"2.0"`:

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

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

However, the occurrence of `^` is an alternative type error location. Instead of string concatenation the programmer may have intended to use an operator on floating point numbers, such as `**` or `+.`:

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

Both programs above are well-typed and may have been intended by the programmer.

Secondly, we observe that a type error message produced by existing functional programming systems is actually very helpful, if it happens to identify the type error location correctly. Existing systems only identify a single type error location; the chosen location depends on the details of the type checking algorithm. For our ill-typed example program the OCaml compiler identifies `2.0` as type error location[2] and adds:

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

This message gives two different types for the type error location `2.0`: Its actual type and an expected type. The expected type is determined by the context of the type error location, the rest of the program. As it is different from the actual type, the expected type is a counter-factual type. The message basically says that if the subexpression at the type error location 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). The programmer still has to select a subexpression of the expected type that meets their intentions. Overall the counter-factual, expected type provides the information to make type debugging easy for the programmer.

Combining our two observations we see how we can simplify type debugging: Instead of giving a type error message with an expected type for just one potential type error location, we want to enumerate such type error messages for all potential type error locations. So for our example we additionally want to obtain a type error message suggesting `^` as the type error location and saying

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

The list of all type error messages with different locations will not be given as a whole to the programmer. Instead the programmer sees one such message at a time, decides whether the type error location has been identified correctly, and only if this is not the case, then the next type error message will be shown. It is clearly desirable to enumerate type error messages such that those that most likely are correct are enumerated early.

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 expected types.

We note that to simplify type debugging, numerous extensions to functional programming systems have been proposed in the literature, but functional programming systems used in practice have hardly changed. 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 for our purpose, but instead describe an algorithm that reuses any existing type checker, which is treated as a black box.

[2]The compiler first outputs the whole let-expression with `2.0` underlined.

In Section 2 we outline our method for enumerating the desired type error messages using any existing type checker. In Section 3 we list the contributions of this paper. In Section 4 we give a formal description for a small core functional language. We describe our implementation in OCaml in Section 5, compare our work with related work in Section 6 and conclude in Section 7.

2 Using 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 Single and Multiple Locations

To simplify the description, we will here only consider searching for a single location that causes a program to be ill-typed. Take for example the ill-typed expression `[1;2;3.]`, which mixes integers and floating point numbers in a list. There are two single potential type error locations:

- 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 `[..;..;..]`[3] may be the type error location. Replacing it by for example a tuple `(..,..,..)` would yield the well-typed expression `(1,2,3.)`.

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.]`. For now we exclude this case. We can extend our method to also handle such cases. Even then it seems sensible to first enumerate all single locations, followed by pairs of locations, etc. It is very likely that the programmer will find the actual cause of the type error early in this list and thus does not have to look at error messages listing several locations.

2.2 Only Leaves as 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, but to debug all type errors it is sufficient to consider only leaf locations. Any program without any leaves is well-typed, because only the use occurrences of variables, data constructors and constants lead to type constraints that cause type errors. It may be possible to debug all type errors by considering only leafs (the replacement may be a complex function that rearranges a whole subtree of the abstract syntax tree), but our method works for both leafs and inner nodes and hence we consider both.

[3]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.

2.3 Determining 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 3 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 3 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.
Error: Here expected an expression of type float -> float -> 'a
```

2.4 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 an expression was expected 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 Contributions

In this section, we describe the advantages of counter-factual type error messages and the contributions of this work.

- Counter-factual type error messages work well for type-annotated program

Programmers often use type annotations to locate the cause of a type error. However, type annotations sometimes do not work as we expect. Let us consider the following program with a type annotation:

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

Although a programmer wrote the program given on the first page, they received the error message shown on the second page. Therefore they added this annotation. However it does not work as expected. OCaml does not treat type annotations specially. OCaml always infers types in a top-down traversal of the program. When OCaml finds the type annotation in this program, it already inferred the type of `f`. Hence OCaml identifies `2.0` as the cause of the type error again. Because counter-factual type error messages are produced independently of the order of type unification, type annotations work more effectively than ever before.

- Our approach is extendable

Using an existing type checker gives us extendability . Our approach requires the ability to insert holes anywhere in a program and to obtain the types of holes. This is not a difficult. We can extend our idea to type definitions, patterns, record definitions and objects. For example, consider the following small program:

```
type st = String of int
let k = String("k")
```

To make a hole in this type definition, we just add a new type variable `'a` that replaces `int`.

```
type 'a st = String of 'a
let k = String("k")
```

We have not yet studied our idea for all features of OCaml's modules, but it definitely works for simple modules. Building a tailor-made type checker for such a big language with so many features would require far more work.

- Our approach does not need to implement a type checker and automatically updates with the compiler

Almost all previous type debuggers use tailor-made type checkers to obtain rich information during inferring types. Although they work well, there are two problems with using tailor-made type checkers. The first problem is implementing the type checker. This is easy for a small language but substantial work for any real programming language such as OCaml or Haskell. Proving that the type checker is correct with respect to the language definition or other compilers is even harder. The second problem is that type systems evolve and change. Some compiler updates provide changed or extended type systems. Such updates require corresponding updates of the tailor-made compiler. Because we use the compiler's own type checker, we avoid these problems.

$$
\begin{array}{rcll}
(\mathtt{M} : \mathit{term}) & ::= & \mathtt{c} & \text{(constant)} \\
 & | & \mathtt{x} & \text{(variable)} \\
 & | & \mathtt{fun\ x\ {->}\ M} & \text{(abstraction)} \\
 & | & \mathtt{@\ M_1\ M_2} & \text{(application)} \\
 & | & \mathtt{(M_1,\ M_2)} & \text{(tuple)} \\
 & | & \mathtt{[M_1;\ M_2]} & \text{(list constructor)} \\
 & | & \mathtt{let\ x\ =\ M_1\ in\ M_2} & \text{(let expression)} \\
(\tau : \mathit{typ}) & ::= & b & \text{(type variable)} \\
 & | & \mathrm{int}, \mathrm{bool}, \ldots & \text{(type constants)} \\
 & | & \tau_1 \to \tau_2 & \text{(function type)}
\end{array}
$$

図 1. The syntax of the let-polymorphic language λ_{let}

4 Obtaining Counter-factual Types

We show our object language in Figure 1. It is the simply-typed lambda calculus extended with tuples, list constructors and let-expressions. We assume that let-expressions have let-polymorphism. The types of this language do not include type schemas. Type schemas are used for inferring polymorphic types during type inference. Because we use an existing compiler's type checker and its inferred types, we do not need to treat type schemas. We assume a compiler's type inference function `INFER` receives an expression and returns its type. If the received expression is not well-typed, it returns an exception `TYPE_ERROR`. (In this paper, we use capitalised `type-writer font` to show that they are the external functions.)

4.1 The Expected Types and Actual Types of Leaves

In this section, we aim to obtain the expected types and the actual types of leaves in the abstract syntax tree. We have already seen the basic idea in Section 2. First, we replace each leaf of the original program by applications of fresh variables. After that, we add the lambda bindings for them at the top of the expression and obtain their expected types and actual types using the compiler's type checker.

Let polymorphism The outlined idea works well if there is no let-polymorphism. To see how to handle let-polymorphism, let us consider the following example:

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

This program is ill-typed. 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 lists(`[1;3;4] :int list` and `[true]: bool list`) to `id`, so the type checker fails. We assume the source of the type error is in `List.iter`. If we replace it by `List.map` (map function: `('a -> 'b) -> 'a list -> 'b list`), then the whole program will be well-typed.

Let us consider to cover `List.iter` with an application of `hole`. Using our original idea, we obtain the following program (Blue parts are changes.):

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

```
    (id [1;3;4], id [true]))
```

However, this transformation is wrong. Although `List.iter` has a polymorphic type in the original program, the replaced expression `(hole List.iter)` has a monomorphic type due to the lambda binding of `hole` (`(fun hole -> ...)`). To make it polymorphic, we move the binding for `hole` under the definition of `id`.

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

By this transformation, the number of `id`'s arguments is increased. 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 the upper program, new variables `hole1` and `hole2` are not bounded. 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 `hole`. From the types of `hole1` and `hole2` we know one of the expected types of `hole` is `('b -> 'b) -> int list -> 'c` and another one is `('d -> 'd) -> bool list -> 'e`. Using these types, we can produce the following 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 an expression was expected of type ('b -> 'b) -> int list -> 'c
         and ('d -> 'd) -> bool list -> 'e
```

The Program We show the functions to obtain the expected types and actual types of leaves in Figure 2. The function `GENSYM` is an external function which produces a fresh string. We assume the names produced by `GENSYM` are unique in the original program. The function `VAR` receives a string and returns a variable using the string. The function `MAP` is the standard map function which applies the received function to each element of the received list.

The function *pierce* transforms expressions to obtain the expected types. It receives an expression and returns a transformed expression and a list of unbounded variables. For variables and constants we introduce holes. We make fresh binders for holes using `GENSYM` and return the transformed expression and the list of fresh variables. In other constructors, we basically call *pierce* recursively to make holes in their sub-expressions. After that, we plug the transformed sub-expressions into the original expressions. In let-expressions, we expand the unbounded variables $[s_1; ..; s_n]$ of transformed M_1 as lambda bindings under the let-expression. Thanks to this process, the holes of the definition part of let-expressions are treated polymorphically.

The function *add* adds the arguments of the target variable. It receives an expression `M`, the name of the target variable f, and a list of its unbounded arguments n. This list n is used to know the number of additional arguments. We assume the size of n is *size_n*. The main work of

$$
\begin{aligned}
&\mathit{pierce} && : && \mathit{term} \rightarrow (\mathit{term} * \mathit{string\ list})\ \mathit{list}\\
&\mathit{pierce}[\![\texttt{c}]\!] && = && \mathit{let}\ \mathit{st} = \texttt{GENSYM()}\ \mathit{in}\ [(\texttt{@}(\texttt{VAR}\ \mathit{st})\texttt{c}, [\mathit{st}])]\\
&\mathit{pierce}[\![\texttt{v}]\!] && = && \mathit{let}\ \mathit{st} = \texttt{GENSYM()}\ \mathit{in}\ [(\texttt{@}(\texttt{VAR}\ \mathit{st})\texttt{v}, [\mathit{st}])]\\
&\mathit{pierce}[\![\texttt{fun x -> M}]\!] && = && \texttt{MAP}\ (\lambda(t,s).(\texttt{fun x -> }t, s))\ \mathit{pierce}[\![\texttt{M}]\!]\\
&\mathit{pierce}[\![\texttt{@ M}_1\ \texttt{M}_2]\!] && = && (\texttt{MAP}\ (\lambda(t,s).(\texttt{@}\ t\ \texttt{M}_2, s))\ \mathit{pierce}[\![\texttt{M}_1]\!])\\
& && && +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{@ M}_1\ t, s))\ \mathit{pierce}[\![\texttt{M}_2]\!])\\
&\mathit{pierce}[\![\texttt{(M}_1\texttt{, M}_2\texttt{)}]\!] && = && (\texttt{MAP}\ (\lambda(t,s).(\texttt{(}t\texttt{, M}_2\texttt{)}, s))\ \mathit{pierce}[\![\texttt{M}_1]\!])\\
& && && +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{(M}_1\texttt{, }t\texttt{)}, s))\ \mathit{pierce}[\![\texttt{M}_2]\!])\\
&\mathit{pierce}[\![\texttt{let x = M}_1\ \texttt{in M}_2]\!] && = && (\texttt{MAP}\ (\lambda(t,[s_1;..;s_n]).\mathit{let}\ (\texttt{M}'_2, s') = \mathit{add}[\![\texttt{M}_2]\!]_{(\texttt{x},[s_1;..;s_n])}\ \mathit{in}\\
& && && \qquad\qquad\qquad (\texttt{let x = fun } s_1\ \texttt{-> ..}\\
& && && \qquad\qquad\qquad\ \ \texttt{fun } s_n\ \texttt{->}\ t\ \texttt{in M}'_2, s'))\ \mathit{pierce}[\![\texttt{M}_1]\!])\\
& && && +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{let x = M}_1\ \texttt{in}\ t, s))\ \mathit{pierce}[\![\texttt{M}_2]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{add} && : && \mathit{term} * \mathit{string} * \mathit{string\ list} \rightarrow \mathit{term} * \mathit{string\ list}\\
&\mathit{add}[\![\texttt{c}]\!]_{(f,n)} && = && (\texttt{c}, [])\\
&\mathit{add}[\![\texttt{v}]\!]_{(f,n)} && = && \mathit{if}\ f = \texttt{v}\ \mathit{then}\ \mathit{let}\ s = \texttt{MAP}\ (\mathit{fun}\ \_ \rightarrow \texttt{GENSYM}())n\ \mathit{in}\\
& && && \qquad\qquad (\texttt{@ v}\ (\texttt{MAP VAR}\ s), s)\\
& && && \mathit{else}\ (\texttt{v}, [])\\
&\mathit{add}[\![\texttt{fun x -> M}]\!]_{(f,n)} && = && \mathit{if}\ \texttt{x} = f\ \mathit{then}\ (\texttt{fun x -> M}, [])\\
& && && \mathit{else}\ \mathit{let}\ (\texttt{M'}, s) = \mathit{add}[\![\texttt{M}]\!]_{(f,n)}\ \mathit{in}\ (\texttt{fun x -> M'}, s)\\
&\mathit{add}[\![\texttt{@ M}_1\ \texttt{M}_2]\!]_{(f,n)} && = && \mathit{let}\ (\texttt{M}'_1, s_1) = \mathit{add}[\![\texttt{M}_1]\!]_{(f,n)}\ \mathit{in}\\
& && && \mathit{let}\ (\texttt{M}'_2, s_2) = \mathit{add}[\![\texttt{M}_2]\!]_{(f,n)}\ \mathit{in}\ (\texttt{@ M}'_1\ \texttt{M}'_2, s_1 + s_2)\\
&\mathit{add}[\![\texttt{(M}_1\texttt{, M}_2\texttt{)}]\!]_{(f,n)} && = && \mathit{let}\ (\texttt{M}'_1, s_1) = \mathit{add}[\![\texttt{M}_1]\!]_{(f,n)}\ \mathit{in}\\
& && && \mathit{let}\ (\texttt{M}'_2, s_2) = \mathit{add}[\![\texttt{M}_2]\!]_{(f,n)}\ \mathit{in}\ (\texttt{(M}'_1\texttt{, M}'_2\texttt{)}, s_1 + s_2)\\
&\mathit{add}[\![\texttt{let x = M}_1\ \texttt{in M}_2]\!]_{(f,n)} && = && \mathit{let}\ (\texttt{M}'_1, s_1) = \mathit{add}[\![\texttt{M}_1]\!]_{(f,n)}\ \mathit{in}\\
& && && \mathit{let}\ (\texttt{M}'_2, s_2) = \mathit{add}[\![\texttt{M}_2]\!]_{(f,n)}\ \mathit{in}\\
& && && (\lambda[].\ (\texttt{let x = M}'_1\ \texttt{in M}'_2, s_2)\\
& && && \quad \mid [s_{11};..;s_{1n}].\mathit{let}(\texttt{M}''_2, s'_2) = \mathit{add}[\![\texttt{M}'_2]\!]_{(x,s_1)}\ \mathit{in}\\
& && && \qquad (\texttt{let x = fun } s_{11}\ \texttt{-> ..}\ \ \texttt{fun } s_{1n}\ \texttt{-> M}'_1\ \texttt{in M}''_2, s'_2)s_1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{infer\_holes} && : && \mathit{term} * \mathit{string\ list} \rightarrow (\mathit{string} * \mathit{typ} * \mathit{typ})\ \mathit{list}\\
&\mathit{infer\_holes}[\![(\texttt{M}, [s_1;...;s_n])]\!] && = && \mathit{try}(\mathit{let}\ ((t_1 \rightarrow t'_1) \rightarrow ... \rightarrow (t_n \rightarrow t'_n) \rightarrow t_{(n+1)}) =\\
& && && \qquad\qquad \texttt{INFER(fun } s_1\ \texttt{-> .. fun } s_n\ \texttt{-> M)}\ \mathit{in}\\
& && && \quad [(s_1, t_1, t'_1); ...; (s_n, t_n, t'_n)])\ \mathit{with}\ \texttt{TYPE\_ERROR} \rightarrow []
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hole\_types} && : && \mathit{term} \rightarrow ((\mathit{string} * \mathit{typ} * \mathit{typ})\ \mathit{list})\ \mathit{list}\\
&\mathit{hole\_types}[\![\texttt{M}]\!] && = && \mathit{let}\ \mathit{lst} = \mathit{pierce}[\![\texttt{M}]\!]\ \mathit{in}\\
& && && \texttt{MAP}\ (\mathit{fun}(\texttt{M}, s) \rightarrow \mathit{infer\_holes}(\texttt{M}, s))\ \mathit{lst}
\end{aligned}
$$

図 2. The functions to obtain the expected types of leaves *add* is performed for the variable case. If the variable is f, *add* replaces it by an application with $size_n$ arguments. In other cases, it simply calls *add* recursively for their sub-expressions. *add* returns the replaced expression and new unbounded variables. For example, we assume the size of n is three. In this case, *add* replaces occurrence of the target variable f in `M` by `@ f` $\mathtt{v}_1$ $\mathtt{v}_2$ $\mathtt{v}_3$ ($\mathtt{v}_n$s are fresh variables) and returns the replaced program and a list of unbounded variables (e.g. `[`$\mathtt{v}_1$`;`$\mathtt{v}_2$`;`$\mathtt{v}_3$`]`).

The function *infer_holes* receives an expression M and its unbounded variables $[s_1; ...; s_n]$. First, we make lambda bindings for the unbounded variables and cover the received expression with them. If the expression is not well-typed, then it will not be a candidate of type error messages. Otherwise, we can obtain its type using the compiler's type checker. Because the type of each unbounded variable has the shape `(its actual type -> its expected type)`, we can separate the types and collect.

The function *main* receives an expression M and returns the names of holes, their actual types and expected types. If the names of holes include the information of the location, we can produce type error messages as follows:

```
# program (underlined using the location information)
Error: This expression has type (its type)
       but an expression was expected of type (expected type)
```

Recursive functions We have to treat recursive let-expressions specially. Let us consider the following example:

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

This program is obviously ill-typed. The variable `x` is of type `string` in `x = "0"` and of type `int` in `x - 1`. We assume that the source of the type error of this program is `"0"`[4] and consider to replace `"0"` by `(hole "0")`. As we have already seen, this `hole` has to have a polymorphic type in `exponential 5`, therefore we add the binder for `hole` under the let-expression. After these transformations, we can obtain the following part-transformed program:

```
let rec exponential = (fun hole -> (fun x -> if x = (hole "0") then 1
                                             else exponential (x - 1))) in
exponential 5 (* This program is halfway through transformation *)
```

The problem is how to handle the recursive call (`exponantial (x - 1)`). Because the type of `hole` must be monomorphic in the definition of `exponential`, we add the same `hole` as the first argument of the recursive call of `exponential` (Red part).

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

Through inferring the type of this transformed program, we can obtain the expected type of `"0"`, `int`.

[4]If we replace it by `0`, the whole program is well-typed.

4.2 The Expected Types of Nodes

In this section, we aim to obtain the expected types of inner nodes of the abstract syntax tree. Let us consider the following example:

```
List.map (fun x -> x + 1) (2, 3)
```

This program is ill-typed, because we pass a tuple `(2, 3)` to `List.map` as the second argument. We assume that the cause of this program is the tuple constructor. If we replace it with a list constructor, then we obtain the following well-typed program:

```
List.map (fun x -> x + 1) [2; 3]
```

To obtain the expected types of nodes, first we consider replacing the focused node with a hole.

```
(fun hole -> (List.map (fun x -> x + 1) hole))
```

This transformed program gives us the expected type of `(2, 3)`, `int list`. To consider the correct program, the types of arguments are also important for programmers. Therefore we pass the arguments of the focused node to the hole.

```
(fun hole -> (List.map (fun x -> x + 1) (hole 2 3)))
```

Altogether we can obtain the expected types of nodes and the types of their arguments.

The naive transformation Some readers may consider the following transformation (It just adds a hole as a function before the focused node).

```
(fun hole -> (List.map (fun x -> x + 1) (hole [2; 3])))
```

This transformation does not work for us, because it does not remove the type restrictions of list constructors. For example, the following simple program is ill-typed:

```
[true; 3]
```

The cause of the type error may be the list constructor (In that case the correct program is `(true, 3)`). If we use the naive transformation, the following transformed program is ill-typed.

```
(fun hole -> hole [true; 3])
```

Every hole that we introduce must remove the type restrictions of the focused part.

Let polymorphism The policy of addition of bindings is identical to the solution in the previous section. If there are holes in the definition parts of let-expressions, we add lambda bindings for them. Otherwise, the unbounded variables are bound at the outermost part of the program as lambda bindings.

The program We show the functions to obtain the expected types of nodes in Figure 3. The function *skeleton* replaces each node by an application of a new variable and their arguments. *skeleton* returns the transformed program and the names of unbound variables. If the target node is under the definitions of let expressions, we add lambda bindings for the unbounded variables. This process is the same with the previous section. The function *infer_nodes* is almost the same with the function *infer_holes* in the previous section. The difference is that *infer_nodes* does not collect the actual type of the focused expression itself. The function *node_types* is also almost the same with the function *hole_types*.

$$
\begin{array}{lcl}
\mathit{skeleton} & : & \mathit{term} \rightarrow (\mathit{term} * \mathit{string\ list})\ \mathit{list} \\
\mathit{skeleton}[\![\texttt{c}]\!] & = & [(\texttt{c}, [])] \\
\mathit{skeleton}[\![\texttt{v}]\!] & = & [(\texttt{v}, [])] \\
\mathit{skeleton}[\![\texttt{fun x -> M}]\!] & = & \texttt{MAP}\ (\lambda(t,s).(\texttt{fun x ->}\ t, s))\ \mathit{skeleton}[\![\texttt{M}]\!] \\
\mathit{skeleton}[\![\texttt{@ M}_1\ \texttt{M}_2]\!] & = & \mathit{let}\ \mathit{st} = \texttt{GENSYM}()\ \mathit{in}\ [(\texttt{@ (var}\ \mathit{st}\texttt{)}\ \texttt{M}_1\ \texttt{M}_2, [\mathit{st}])] \\
 & & +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{@}\ t\ \texttt{M}_2, s))\ \mathit{skeleton}[\![\texttt{M}_1]\!]) \\
 & & +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{@ M}_1\ t, s))\ \mathit{skeleton}[\![\texttt{M}_2]\!]) \\
\mathit{skeleton}[\![\texttt{(M}_1\texttt{, M}_2\texttt{)}]\!] & = & \mathit{let}\ \mathit{st} = \texttt{GENSYM}()\ \mathit{in}\ [(\texttt{@ (var}\ \mathit{st}\texttt{)}\ \texttt{M}_1\ \texttt{M}_2, [\mathit{st}])] \\
 & & +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{(}t\texttt{, M}_2\texttt{)}, s))\ \mathit{skeleton}[\![\texttt{M}_1]\!]) \\
 & & +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{(M}_1\texttt{,}\ t\texttt{)}, s))\ \mathit{skeleton}[\![\texttt{M}_2]\!]) \\
\mathit{skeleton}[\![\texttt{[M}_1\texttt{; M}_2\texttt{]}]\!] & = & \text{almost the same with the tuple case} \\
\mathit{skeleton}[\![\texttt{let x = M}_1\ \texttt{in M}_2]\!] & = & (\texttt{MAP}\ (\lambda(t,s).\mathit{let}\ (\texttt{M}'_2, s') = \mathit{add}[\![\texttt{M}_2]\!]_{(x,s)}\ \mathit{in} \\
 & & \qquad (\texttt{let x = fun s -> t in M}'_2, s'))\ \mathit{skeleton}[\![\texttt{M}_1]\!]) \\
 & & +\ (\texttt{MAP}\ (\lambda(t,s).(\texttt{let x = M}_1\ \texttt{in}\ t, s))\ \mathit{skeleton}[\![\texttt{M}_2]\!]) \\
 & & \\
\mathit{infer\_nodes} & : & (\mathit{term} * \mathit{string\ list}) \rightarrow (\mathit{string} * \mathit{typ})\ \mathit{list} \\
\mathit{infer\_nodes}[\![(\texttt{M}, [s_1; ...; s_n])]\!] & = & \mathit{try}(\mathit{let}\ t_1 \rightarrow ... \rightarrow t_n \rightarrow t_{(n+1)} = \\
 & & \qquad \texttt{INFER(fun s}_1\ \texttt{-> ... fun s}_n\ \texttt{-> M)}\ \mathit{in} \\
 & & \quad [(s_1, t_1); ...; (s_n, t_n)])\ \mathit{with}\ \texttt{Type\_Error} \rightarrow [] \\
 & & \\
\mathit{node\_types} & : & \mathit{term} \rightarrow ((\mathit{string} * \mathit{typ})\ \mathit{list})\ \mathit{list} \\
\mathit{node\_types}[\![\texttt{M}]\!] & = & \mathit{let}\ \mathit{lst} = \mathit{skeleton}[\![\texttt{M}]\!]\ \mathit{in} \\
 & & \texttt{MAP}\ \mathit{infer\_nodes}\ \mathit{lst}
\end{array}
$$

図 3. The functions to obtain the expected types of nodes

4.3 A Study of type error messages's Ordering

We introduced an approach to make type error messages of all single potential type error locations using an existing compiler's type checker. Because we might sometimes have many messages for an ill-typed program, the messages's order of showing programmers is important. If we simply show all type error messages, the needed time to locate the source of the type error depends on the order.

To define an efficient order, we need experiences, such as many ill-typed programs and their sources of the type errors. Frequent mistakes may be differ depending on the programmers. Therefore, we need a way to gather the experiences and to extract the information about possibility of the source of the type error from them automatically. In this paper, we do not go further about this problem.

4.4 Interactive Type Debugging

In this section, we consider the possibility about interactive type debugging using our approach. First, we consider the program that we introduced in Section 1 again.

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

We can obtain 4 different type error messages [5] from this program. They are about `f` (in `f 2.0`), `2.0` (in `f 2.0`), `^` and `n` (in `x ^ n`). First, we consider the case that the first error message is about `n` as the following:

```
# let f = (fun n -> (fun 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
```

Now, we assume the programmer's intended source of the type error is in `^`. In this case, this error message is inept. Because the actual type is the programmer's intended type, the programmer answers this expected type, `string`, is not her/his intended type. Therefore, we can memorize the information that programmer's intended type of `n` (in `x ^ n`) is not `string`. Using this information, we remove some candidates of the type error messages.

To judge which candidates we can remove, we just make another hole, the same hole with the upper message in the candidates. For example, we consider the following candidate about `2.0`:

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

We make a hole at the part of `n` in this program.

```
(fun hole2 -> (fun hole ->
let f = (fun hole2 ->
   (fun n -> (fun lst -> List.map (fun x -> x ^ (hole2 n)) lst))) in
f hole2 (hole 2.0)))
```

Using this transformed program, we can obtain the expected type of `n`, `string`. Because this has a conflict with the previous programmer's answer, we can remove this type error message from the candidates. Although this is the case two types are the same, the programmer's answers work if they are the same. To judge two types have type conflicts, we can use the function which determines "more general" [6]. It receives two types, A and B, and determines if A is more general than B. For example, we have two types `string -> 'a` and `'b -> 'c`. The latter is more general than the former. If a programmer's previous answers of a focused expression are more general than its inferred type, we can judge the candidate is not needed.

In a similar way, we can refresh candidates using the programmer's answers. In this case, we have only one candidate `^` after asking the first error message. Like this, although we have many candidates first, the programmer do not have to see all of them.

4.5 A Study of Several Type Errors in an Ill-typed Program

An ill-typed program often includes many type conflicts. In this section, we describe how to deal them in our approach.

Let us consider the following example which may include some causes of type errors:

```
[1; 2.; "3"]
```

[5]Actually, we obtain 6 type error messages using the functions we show in Figures 2 and 3. Although they include `f 2.0` (application) and `x ^ n` (application), their transformed programs are the same with `f` (in `f 2.0`) and `2.0` (in `f 2.0`) respectively. Therefore we omit 2 messages here.

[6]In OCaml, there is a function `Ctype.moregeneral`.

Because a list constructor requires its all elements to have the same type, this program is ill-typed. Here, we assume the programmer considers this program should have type `float list` [7], therefore the causes of the type errors are in `1` and `true`. First, we obtain a candidate about the list constructor using the functions in Sections 4.1 and 4.2. The error message about the list constructor is the following:

```
# [1; 2.; "3"]
Error: This expression's arguments have types int, float and bool respectively
       and its expected type is 'a,
       but this constructor requires the type 'b -> 'b -> 'b -> 'b list.[8]
```

However, this constructor is not the cause of the type error for the programmer. At this point, we already show all candidates because there is only one single potential type error location in the original program. Therefore, we reconstruct a new program from the original ill-typed program. Because we know the list constructor is the programmer's intended one, we add it first and after that we add other parts until the whole program will be ill-typed. For example, in this case, we can obtain a new ill-typed program `[1; 2.; □]`[9]. In this program, `1` and `2.` cause a type conflict. Because `"3"` was abstracted, `1` and `2.` become candidates for single type error locations in this program. One message of the candidates (about `1`) is the following:

```
# [1; 2.; "3"][10]
Error: This expression has type int
       but an expression was expected of type float.
```

Because we assume the programmer's intended type of the whole program is `float list`, this error message shows one of the sources of the type errors in the original program. After s/he corrects it, we restart type debugging to locate another source of the type error, `"3"`.

As just described, if an ill-typed program has several sources of the type errors, we can debug it by repeating debugging and narrowing of the original program. This narrowing is done by a kind of type error slicing[11].

[7]For example, `[1.; 2.; 3.]` might be the correct intended program.

[8]The upper two lines are about the expected types, and the last line is about the actual type of the focused list constructor. In Section 4.2, we do not touch about the actual type of nodes. We can obtain them by replacing the sub-expressions of the focused expression by fresh variables and bind them. In this case, we make `(fun a -> fun b -> fun c -> [a;b;c])` from the focused expression and obtain the type `('b -> 'b -> 'b -> 'b list)` using a compiler's type checker.

[9]This □ is a kind of holes, however we do not take notice of its type.

[10]Although we show the original program in this message,
type debugger uses different program `[1; 2.; _]`

[11]Usually the aim of type error slicing is to obtain the minimum slice from the original program. We do not try to obtain the minimum slice here.

5 Implementation in OCaml

We have implemented a prototype of our approach for a subset of OCaml 3.12.1. Our target language includes standard expressions, such as if-expressions, constructors and the like. Thanks to the following features of OCaml, we can minimize our effort of implementation.

- the abstract syntax tree for expressions and types
- the lexer, the parser, and pretty printer for types
- the type checker `infer` (that receives an expression and returns its type or an exception `Type_Error`)

Because we utilize the lexer and parser of OCaml, our work is transforming the abstract syntax tree by the functions shown in Figures 2 and 3. Each part of the OCaml's abstract syntax tree have the information of its program location. Using these information we generate new symbols (`GENSYM`) including its location information.

We do not change these compiler's functions at all. If we extend the object language with modules, objects and so on, we will use these strongly typed language's fundamental features as is. Therefore it will be easy to implement our approach in other languages.

To improve the behavior of our implementation, we also use the following features of OCaml.

- the function `moregeneral` (that determines if one type is more general than the other type) and the type unifier `unify` (that receives two types and returns the unified type)

We use the function `moregeneral` to realize an interactive type debugging and `unify` to realize recommendations of amendments.

Recommendations of amendments We have the expected types in our type error messages. If the expected types are the programmer's intended one, we can make recommendation of amendments using them. For example, let us consider the program shown in Section 4.1.

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

We assume the cause of this type error is in `List.iter`. If we replace it by `List.map`, the whole program will be well-typed. The error message about `List.iter` is the following:

```
# 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 an expression was expected of type ('b -> 'b) -> int list -> 'c
       and ('d -> 'd) -> bool list -> 'e
```

From this message, we know the expected type of `List.iter` is `('b -> 'b) -> int list -> 'c` and `('d -> 'd) -> bool list -> 'e`. Because we know the type of `List.map` is `('f -> 'g) -> f list -> 'g list`, we can judge the expected types and the type of `List.map` do not have type conflicts. Therefore we can recommend `List.map` as an amendment. To produce recommendation of amendments, we need a list of candidates of amendments. We judge which ones do not have type conflicts with the expected types using the function `unify`. In our implementation, we use the functions in OCaml's initially opened module as the candidates of amendments.

Although the amendments are often useful for programmers, there are two problems. First, amendments can not always show the correct fix for programmers's intentions. Second, they have risks to cause incorrect fixes. To recommend of amendments practically, we should take up these problems.

6 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. 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 [10] 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 [6] 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. Heeren and Hage [5] use a constraint-based type checker to flexibly vary the order of solving constraints. Their heuristic approach sometimes but not always produces good error messages. In contrast, our approach is completely independent of the order of solving type constraints.

Reusing an Existing Type Checker

Braßel[1] was the first to proposes using an existing type checker for type debugging. His experimental tool TypeHope automatically corrects type errors for the functional logic language Curry. Lerner et al. [7] take this idea further. They replace the erroneous part with various syntactically correct similar expressions, and see if they type check. If they do, they are displayed as the 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 Lerner et al. [7] but like Chen and Erwig [2], our aim is to suggest type changes and leave it to the programmer to select the appropriate expressions.

Counter-Factual Types

Chen and Erwig [2] first proposed to assist type debugging by automatically enumerating 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 [3] developed an algorithmic debugger for type debugging, using a compositional type inference algorithm. Based on his work, Tsushima and Asai [9] 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. However, answering the questions of an algorithmic debugger requires a good understanding of types, especially intended types, by the programmer. Our approach of enumerating counter-factual type error messages is easier to use.

Type Error Slicing

The idea of type error slicing is to determine for a type error a small slice of the program which contains all the program parts responsible for the type error. To correct the type error, a program change within the slice is required. Haack and Wells [4] define and implement type error slicing for ML using their own type checking algorithm which solves annotated type constraints. Later Schilling [8] obtains type error slices for a large subset of Haskell using the compiler's type checker as a black box. The advantage of type error slicing is that the process is fully automatic and the programmer does not have to answer any questions. A disadvantage is that even minimal slices can be still 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 related. All the potential locations identified by our approach together form a type error slice. When a program contains several type errors, we search for potential error locations similar to type error slicing (cf. Section 4.5).

7 Conclusion and Future Work

In this section, we fleshed out our thesis that we can produce type error messages about all single potential type error locations using a compiler's type checker. If the program includes polymorphic errors, our approach produces enough good error messages to locate the causes of the type error. Our idea is very simple; we make hole(s) in the program and obtain their type(s). We have illustrated the thesis with simply-typed lambda calculus expanded with let-expressions. We show the possibility of interactive type debugging using our approach and handling of the programs including several type errors.

Our future work is twofold. First, we want to extend our approach and implementation to apply many constructors in real languages. Because we confirmed that we could extend our approach with patterns, objects and etc., our work about these advanced features is just to implement it. We want to confirm about the other features, such as GADTs, modules, and etc.. At this time there are a problem about bindings. Because the elimination of bindings will cause unbounded variables, we did not treat them as candidates of the causes of the type errors in this paper. However, they may sometimes be the source of the type error. Therefore we need a way to treat them as candidates but avoid the problem of unbounded variables. Second, we want to check our approach for the viewpoints of scalability for large programs and usability. About scalability, our implementation works for 7 lines program, it infers types for about 30 transformed programs and finds 3 candidates soon. If we find a candidates, we can use the programmer's thinking time for searching the other candidates. To confirm whether our approach scales in many examples, it will need an implementation with advanced features and users of our implementation. The user tests will give us a direction for improvements of this work.

参考文献

[1] Braßel, B. "Typehope: There is hope for your type errors," *15th International Workshop on Implementation of Functional Languages (IFL'04)*.

[2] Chen, S., M. Erwig. "Counter-Factual Typing for Debugging Type Errors," *Proceedings of the 41th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'14)*, to appear (2014).

[3] Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," *Proceedings of the sixth ACM JOPLIN international conference on Functional programming (ICFP'01)*, pp. 193–204 (2001).

[4] 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).

[5] Heeren, B., J. Hage. "Parametric Type Inferencing for Helium," Technical Report UU-CS-2002-035, Utrecht University, 2002.

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

[7] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. "Searching for Type-Error Messages," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pp. 425–434 (2007).

[8] Schilling, T. "Constraint Free Type Error Slicing," *Proceedings of the 12th international conference on Trends in Functional Programming (TFP'11)*, pp. 1–16 (2012).

[9] 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).

[10] Wand, M. "Finding the Source of Type Errors," *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'86)*, pp. 38–43 (1986).