Report on a User Test and Extension of a Type Debugger for Novice Programmers

Yuki Ishii
Department of Information Science,
Ochanomizu University
Tokyo, Japan
ishii.yuki@is.ocha.ac.jp

Kenichi Asai
Department of Information Science,
Ochanomizu Univeristy
Tokyo, Japan
asai@is.ocha.ac.jp

A type debugger interactively detects the expressions that cause type errors. It asks users whether they intend the types of identifiers to be those that the compiler inferred. However, it seems that novice programmers often get in trouble when they think about how to fix type errors by reading the messages given by the type debugger. In this paper, we analyze the user tests of a type debugger and report problems of the current type debugger. We then extend the type debugger to address these problems. Specifically, we introduce expression-specific error messages and language levels. Finally, we show type errors that we think are difficult to explain to novice programmers. The subjects of the user tests were 40 novice students belonging to the department of information science at Ochanomizu University.

1 Introduction

Strongly-typed languages, such as OCaml or Haskell, provide programmers with type safety via static type checking at compile time. However, it is not always easy for programmers, especially novices, to write well-typed programs. In particular, the error messages provided by the compiler often do not indicate the source of the type error. For example, the OCaml compiler prints the following message, when a programmer tries to define a function that calculates the `x`-th power of `(x + 1)`:

```
fun x -> (x + 1) ^ x
Error: This expression has type int
       but an expression was expected of type string
```

This error message says that the type of `x + 1` conflicts with the type of the first argument of `^`, which concatenates two string values in OCaml. If the programmer blindly follows the error message and changes the type of `x + 1` to `string` (e.g., by inserting `string_of_int`), he or she will end up with a different program than originally intended:

```
fun x -> (string_of_int x + 1) ^ (string_of_int x)
```

Since the compiler does not know the intention of the programmer, it is unable to show a single error message that reflects that intention. The above error message simply reports the fact that the two types are in conflict during type inference. To remedy this situation, Chitil [1] proposed an interactive type debugger. Using algorithmic program debugging [13] on the compositional type inference tree, this sort of debugger ascertains the programmer's intention by asking questions and detecting the sources of the type errors. Tsushima and Asai [16] followed up on this work by implementing a type debugger for full OCaml by reusing its type inferencer.

In 2012, we used the OCaml type debugger in a "Functional Programming" course at Ochanomizu University and collected logs showing how students interacted with the type debugger. In this paper, we

J. Caldwell, Ph.K.F. Hölzenspies, P. Achten:
Trends in Functional Programming in Education (TFPIE 2014)
EPTCS 170, 2014, pp. 1–18, doi:10.4204/EPTCS.170.1

report on the results of analyzing the logs, describe problems of the type debugger, and extend the type debugger accordingly to make it novice-friendly.

This paper is structured as follows. In Section 2, we review how our type debugger works. In Section 3, we analyze the logs of the type debugger and discuss the results in Section 4. We extend the type debugger in Section 5 and evaluate it in Section 6. Its limitations are discussed in Section 7. Related work is in Section 8, and we conclude the paper by outlining future work in Section 9.

2 Type debugger

Let us review how the type debugger works. The type debugger constructs the most general type tree [1, 16] and uses algorithmic program debugging [13] to detect type errors.

2.1 Most General Type Tree

The type debugger uses the most general type tree (MGTT) to detect the source of a type error. Unlike the standard type inference tree used in a compiler, an MGTT maintains the most general type for each subexpression. For example, the MGTT for the previous program becomes as follows: (Here, we have abbreviated the types of `+` and `^` as $\tau^{+} = \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$ and $\tau^{\wedge} = \texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{string}$, respectively.)

$$\dfrac{\dfrac{\dfrac{\boxed{\{\texttt{x : a}\}}^{A} \vdash \texttt{x : a} \qquad \boxed{\{\}}^{A} \vdash \texttt{+} : \tau^{+} \qquad \boxed{\{\}}^{A} \vdash \texttt{1 : int}}{\boxed{\{\texttt{x : int}\}}^{B} \vdash \texttt{(x + 1) : int}} \qquad \{\} \vdash \text{\textasciicircum} : \tau^{\wedge} \qquad \{\texttt{x : b}\} \vdash \texttt{x : b}}{\{\} \vdash \texttt{(x + 1)}\ \text{\textasciicircum}\ \texttt{x} \cdots \texttt{type error}}}{\texttt{fun x} \rightarrow \texttt{(x + 1)}\ \text{\textasciicircum}\ \texttt{x} \cdots \texttt{type error}}$$

The MGTT is different from the standard type inference tree in that the information that `x` has type `int` in box *B* does not propagate to boxes *A*. To type the three subexpressions, `x`, `+`, and `1`, independently, we do not need to constrain the type of `x`. The type of `x` becomes `int` only when these subexpressions are composed and `x` is passed as an argument to `+`. This kind of bottom-up type inference was used in $\mathscr{U}_{AE}$ [18] as well as in the Helium compiler [6] to remove the left-to-right bias of the type inference and produce better error messages.

MGTT is compositional: the most general type of an expression is determined solely from the expression and does not depend on other expressions. By comparing the most general type with the programmer's intended type, we can detect the source of a type error [1]. In this paper, we say that an expression has a <u>well-intended</u> type if the type of the expression does not contradict the programmer's intention.

For example, suppose that a programmer intends the above function to have one of two types:

1. `int -> string`
2. `int -> int`

In the first case, the programmer's intended program is `fun x -> string_of_int (x + 1) ^ string_of_int x`. While the type of `x` in boxes *A* does not contradict the programmer's intention, the type of `x` in box *B* does contradict them. In other words, `x`, `+`, and `1` all have the well-intended types, but `(x + 1)` does not. Thus, we conclude that `x + 1` is the source of the type error.

If we used the standard type inference tree instead, we could detect that the conflict first occurs somewhere in boxes *A* or *B*. However, we cannot further identify the source of the type error, because the type of `x` propagates to the boxes *A* via unification and we have no information when the type of `x` is first forced to `int`.

On the other hand, suppose that the programmer intends the second type. In this case, the intended program is `fun x -> power (x + 1) x` where `power x y` calculates the `x`-th power of `y`. Since the programmer intends `^` to be `int -> int -> int`, the actual type of `^` conflicts with the programmer's intention. Therefore, we can detect that `^` is the source of the type error.

2.2 Algorithmic Programming Debugging

The type debugger detects the source of a type error by using algorithmic program debugging (APD) to traverse the MGTT [13]. APD was originally devised by Shapiro to find an error in a Prolog program. It can be used to detect errors in any tree structure. The algorithm starts from a node with an error and proceeds as follows.

1. Check whether any of its child nodes has an error.
2. If no child node has an error, the current node is the source of the error.
3. If one of the child nodes has an error, apply APD to the child node.

In the last step, if two or more child nodes have an error, one of the erroneous child nodes is chosen. The final result depends on which one is chosen.

2.3 Detecting type errors

Our type debugger detects the source of a type error in two steps:

1. Find the node (expression) that has a type error, but all of its child nodes are well-typed.
2. Find the node (expression) that does not have well-intended types, but all of its child nodes have well-intended types.

In the first step, the type debugger uses the compiler's type inferencer to judge whether a node has a type error. In the second step, the type debugger asks the programmer whether the types inferred by the compiler's type inferencer match his/her intention. In particular, it asks whether the environments and expressions are of the intended types.

In the MGTT in Section 2.1, for example, the type debugger starts from the bottom of the tree and reaches node `(x + 1) ^ x` as a result of executing step 1, because all of its child nodes are well typed. Starting from this node, the type debugger asks the programmer if each subexpression has the intended type.

If the programmer answers that all the child nodes have intended types, the current expression is identified as the source of the type error, because the type of `x + 1` does not match the type of the first argument of `^`. In this case, the identified expression is ill-typed.

On the other hand, if the programmer answers that the type of `x` should be `string` (in box *B*), the node `x + 1` is identified as the source of the type error, because the type of `x` is first forced to `int` here. In this case, the identified expression is well-typed, but not well intended.

expression	%
Application	30.2
Match expression	13.5
Constructor	11.4
Conditional	4.3
Recursive function	3.7
Environment	18.7
Syntax misunderstood	18.2

Table 1: Classification by expression

3 Analysis

In the spring of 2012, we used the type debugger in a "Functional Programming" course offered at our university. The course was taken by 40 CS-major students. Although they had one-year of experience writing C programs, it was the first time for them to write programs in a strongly-typed language. During the course, the students learned OCaml and wrote a solution to the shortest path problem for the Tokyo metro network. We instructed students to use the type debugger when they encountered type errors. When (and only when) they used the type debugger, we collected the erroneous programs and their interactions with the type debugger.

We analyzed the type-error logs in two ways:

1. Which expression was detected as the source of the type error?
2. How did the students change their programs after reading the error message?

3.1 Expressions identified as sources of type errors

Table 1 shows a breakdown of expressions identified as sources of type errors by the type debugger. We collected 704 logs and classified them manually. Among the seven categories, the identified expression was ill-typed in the first five categories and well-typed in the last two categories.

In this section, we describe typical type errors from the logs and analyze them to see if the type debugger worked effectively.

Application. 30.2% of the sources of type errors were located in the application. After an application expression was identified, the type debugger determined which of the arguments caused the type error. It passed an increasing number of arguments, starting from the first one, to the function. The first argument that caused a type error was shown to the programmer as the conflicting argument.

For example, a typical type error is shown below. The box in the program shows the highlighted part.

```
fun x -> [(x + 1) ^ x]1
Error:
The first argument of this application causes a type error. (highlight 1)
```

In the error message, "`this application`" refers to the application of the `^` operator and "`the first argument`" refers to `x + 1`. However, since the `^` operator uses infix notation, most students had trouble understanding which expression "`this application`" refers to.

Moreover, even after students understood what "`this application`" refers to, the error message was still not very helpful:

```
(* f : int list -> int -> int list *)
(* g : int list -> int list *)
let test = [f (g lst) = [a; very; large; list;...]]1
Error:
The second argument of this application causes a type error.
```
(highlight 1)

Although this error message suggests that "`[a; very; large; list; ...]`" is the source of the type error, it is actually not. What caused this type error is that the student passed only one argument to function `f`, which required two. It resulted in the type of = being `(int -> int list) -> (int -> int list) -> bool`.

Since the error message mentions only the second argument, the students tended to check only the large list and rarely found they had forgotten an argument for `f (g lst)`. Even if the student could understand the error message, it seemed to be of little help in finding and fixing the source of the type error.

Match expression. As the course progressed, the programs that the students wrote became more complicated. One sort of the complex expressions that the students wrote is match expressions. Here is an example where a student struggled to correct a type error more than ten times.

```
type station_t = {start : string; destination : string; distance : float;}
type tree_t = Empty | Node of tree_t * string * (string * float) list * tree_t
let rec insert_station station_tree station =
  match station with                                                        3
   [] -> [[]]1
   |{start = st; destination = dest; distance = dist;} :: rest ->
      match station_tree with                                               2
        Empty -> Node (Empty, st, [(dest, dist)],Empty)
        | Node (t1, name, station_list, t2) ->
            if name < st then
              Node (t1, name, station_list, insert_station t2 station)
            else if name > st then
              Node (insert_station t1 station, name, station_list, t2)
                else insert_name station_tree rest
Error:
This match expression causes a type error.
```
(highlight 3)

The type error occurred because the match expression (in highlight 3) returns an empty list in highlight 1 but a tree in highlight 2. However, after asking various questions, the type debugger identified the whole match expression (highlight 3) as the source of the type error and responded with an uninformative error message. Because a large area (highlight 3) was identified as the source of the type error, the student had a hard time finding out which part of the highlight was wrong.

The source of the type error was identified as the whole match expression, because the type debugger works at the level of expressions. It first checks whether all the subexpressions are well-typed: i.e., it checks whether the expression to be matched (`station`) as well as all the branches of the match expression are well-typed. When the student answers that their types are as intended, the debugger concludes that type constraints of this match expression are not satisfied. Working at the level of expressions has the benefit that we can build a type debugger without adding any expression-specific type constraints. If we had to consider all the type constraints for all the expressions, it would have been difficult to support all the constructs in OCaml. However, the fact that the students had to struggle debugging the above program shows that we should actually consider expression-specific type constraints for common cases so that we can provide more informative error messages.

Conditional expression. The same problem applies to conditional expressions. Whenever a conditional expression is identified as the source of a type error, the type debugger shows the same error message. However, a conditional expression has its own type constraints: the predicate part has to have type `bool` and the two branches must have the same type. In these cases, it would be more informative to show them in the error message.

Another common mistake regarding conditional expressions happens when students forget to write the `else` branch. In this case, the `then` branch must have the `unit` type. Even if students could answer the questions asked by the type debugger correctly, they might still be puzzled by the error message, saying that the `then` branch must have the `unit` type, because the `unit` type is introduced in a later stage of the course; they would not know what a `unit` is at this point.

Constructor. When students define a new data type, the types of the arguments are often wrong. For example, suppose we have a tree with `char` and `int` information at each node:

```
type tree_t = Empty
            | Node of tree_t * char * int * tree_t
```

If a student forgets to write one of its argument, e.g., `Node (left, n, right)`, the type debugger identifies this expression to be the source of the type error: after confirming that the student intended the tuple to have the type `tree_t * int * tree_t`, it reports that the number of arguments of `Node` is four. However, since the definition of `Node` is usually placed far before its use, the students would not immediately see why four arguments instead of three are required.

Because constructors are not functions and are handled differently from functions in OCaml, the current implementation of the type debugger does not ask the intended type of constructors, but rather assumes that the types are as intended. The above problem could have been avoided if the type debugger asked the intended type of the constructor and jumped back to its definition when it was different from the inferred type.

Recursive functions. In some cases, students try to use a recursive function of a different type from its definition. In such case, the current error message shows the two types of recursive function: the type inferred from the outermost structure and the type of its use. For example, for the following function where the second argument to `gcd` is missing at the recursive call:

```
let rec gcd [m n = if n = 0 then m else gcd n] 1
```

the type debugger shows the following error message:

```
While this expression has 'a -> int -> 'a,
you tried to use it as type int -> 'a for recursive call.  (highlight 1)
```

	Effective	appropriate expression, but ineffective	unrelated expression	No changes
Application	36.3	18.2	12.5	33.0
Match expression	38.0	10.3	17.2	34.5
Constructor	20.7	17.2	6.9	55.2
Conditional	0	50.0	0	50.0
Recursive function	7.7	7.7	23.1	61.5

Table 2: Analysis by reaction of students (%)

Well-typed but unintended expressions. If a student answered that the type shown in the MGTT is different from his or her intention, the type debugger traverses the MGTT to the node whose expression is well-typed. This case can be divided into two subcases.

One subcase is when the type of a variable in the environment is not the intended one. This situation is listed in Table 1 as "Environment". Here, the first node where the type of the variable is forced as such is identified as the source of the type error. From that, students can understand why the variable's type is not as intended.

The other subcase is when students misunderstand (or mistakenly use) syntax. This situation is listed in Table 1 as "Syntax misunderstood". For example, if a student writes a floating point number 1.2 as `1,2`, it is well-typed as `int * int`, but not as `float`. Another example is to write a list of numbers `[1; 2]` using a comma `[1, 2]`. The latter is a singleton list of a pair of integers. In these cases, the type debugger shows the type of the expression, which is enough for students to understand the cause of the type error most of the time.

3.2 Students' reactions

For the first five cases of Table 1, we classified how the students changed their program after they had read an error message from the type debugger in four ways, following Marceau et al. [9]:

1. An effective change that fixes the type error and brings the student closer to a solution.
2. An ineffective change to the appropriate expression.
3. A change to an expression that is unrelated to the type error.
4. No changes, such as inserting indentation or spaces.

We can judge the effectiveness of changes, because we know what the student is aiming for: the solutions to the assignments. In addition, we use the purpose statement the student writes to guess his intention. In the course, students were advised to follow the design recipe [3] and write a purpose statement together with its type for each function definition. When we classified the students' reactions, we used this information to judge if the change was effective. However, the type they write in the purpose statement is often wrong.

Table 2 shows the classification of the students' reactions. We found that not many programs could be corrected effectively. Although students could fix nearly 40% of the type errors in applications or match expressions, they could not make any changes to a third of them. Furthermore, more than 60% of the type errors located in recursive functions were left unchanged.

We analyzed the application, match expressions, and conditional expressions as follows.

Application. When an application is detected as the source of a type error, the type debugger prints the argument that caused the type error. As a result, the students often changed that argument or swapped the arguments. However, they seemed to get in trouble when the infix notation was used, or the argument was unintentionally higher-order (such as we saw in Section 3.1).

Match expression. Students tended to change the wrong expression when the highlight covered a wide area of the program. Since match expressions often include complicated data types, students sometimes changed those complicated expressions to simpler ones without type errors and the students could identify where to fix them. This accounted for 38%, a relatively high percentage, of the effective changes.

Conditional expression. As we saw in Section 3.1, students tended to get in trouble when they forgot to write `else` statements. In this case, they often changed the `then` statement into something else and rarely noticed that what they needed to do was add an `else` statement.

4 Discussion

The analysis in Section 3 shows that there is plenty of room for improvement when it comes to helping novice programmers use the type debugger effectively. Our main goal here is to help students understand type errors and learn how to program in typed languages in general, by reading error messages of the type debugger. In this section, we discuss the problems of the current type debugger as regards this goal.

The first problem is that the type debugger does not show enough information for programmers to understand why the type errors occurred. As the second program of application in Section 3.1 shows, students could not find the actual source of the type error in the first argument, because the type debugger said the second argument caused the type error. Thus, rather than simply showing the first type-conflicting argument, we should show more information, such as the types of other arguments, so that students can consider what went wrong.

The second problem is that the detected expressions are sometimes too large for students to find the source of the type error. If the detected expression is ill-typed and violates a type constraint, the type debugger should show it so that students can see immediately what constraint is violated.

In light of these results and considerations, we decided to extend the type debugger in three ways.

1. Smaller highlights:
   When ill-typed conditional or match expressions are detected as the source of the type error, we use expression-specific type constraints to narrow the highlighted part. This also leads to novice-friendly error messages.

2. Novice-friendly error messages:
   In addition to expression-specific type error messages, we provide detailed information why the type error occurred, in particular for applications.

3. Introduction of language levels [2] (as in Racket[1]):
   To provide appropriate error messages, we prohibit the use of advanced language features that students are not supposed to use.

We describe these extensions in the next section.

[1] http://racket-lang.org/

5 Extensions

5.1 Expression-specific error messages

As we saw in Section 2, the first step of type debugging detects a node in a tree that is ill-typed but whose child nodes are all well typed. If types of all its subexpressions are as intended, the type debugger reports the current expression as the source of the type error. When the detected expression itself is ill-typed, however, we want to have a detailed explanation as to why it is ill-typed.

To provide programmers with more informative error messages, we introduce expression-specific error messages for conditional expressions, match expressions, and applications. The basic idea is to add annotations to the subexpressions of the ill-typed expression and see if they satisfy the type constraints imposed by the expression.

5.1.1 Conditional expressions

The type constraints for conditional expressions are:

1. The predicate part must be of type `bool`.
2. When the `else` branch is missing, the `then` branch must be of type `unit`.
3. When the `else` branch is present, the `then` and `else` branches must have the same type.

To examine if these constraints are satisfied, the type debugger is extended to perform the following checks:

1. The type debugger extracts the predicate part of the conditional expression and annotates it as `bool`. It then passes the annotated predicate to the compiler's type inferencer. If it reports a type error, the type debugger identifies the predicate part as the source of the type error.
2. When the `else` branch is missing, the type debugger extracts the `then` branch and annotates it as `unit`. It then checks if the annotated branch type checks as in 1. If it does not, the type debugger identifies the `then` branch as the source of the type error.
3. When the `else` branch is present, the type debugger constructs a list of the `then` and `else` branches and checks if it type checks. If it does not, the type debugger identifies that the type error occurred because the two branches had different types.

With this extension, the type debugger can provide better error messages for conditional expressions. For example, before the extension, in all the following cases taken from the logs, the type debugger reported the same error message, saying only that the conditional expression was the source of the type error. The newly generated messages look as follows.

1. Predicate:

   ```
   ... if [(try kekka_ kyori with Not_ found -> infinity)]1 then ...
   Error:
   The type of predicate statement is float, but it should be bool.
   ```
   (highlight 1)

2. Without `else` statement:

   ```
   ... if (a + 1) < c then [a + 1]1
   Error:
   The type of then statement is int but it should be unit.
   ```
   (highlight 1)

3. With `else` statement:

```
... if (a + 1) < c then a + 1 else print_int c
```
[1]

```
Error:
The type of then statement is int and else statement is unit,
but these should be the same type.
```
(highlight 1)

In the error messages, the type debugger prints the types of related subprograms and how the programmers need to change it. Also, in the first two examples, the debugger uses smaller highlights, where only detected subprograms are colored.

5.1.2 Match expressions

The type constraints for match expressions are:

1. The expression to be matched and all the patterns must have the same type.
2. All the branches must have the same type.

We extended the type debugger to examine these constraints, as well as the conditional expressions.

1. The type debugger replaces all the branches with a dummy value (such as `()`) and checks if the resulting match expression is well-typed. If it is not well-typed, the type debugger identifies the first pattern that causes a type error by repeatedly removing the last pattern-expression pair from the list. If the first pattern already causes a type error, it means that the expression to be matched and the first pattern do not have the same type.
2. If all the patterns have the same type as the expression to be matched, the type debugger performs the same things for the original match expression: it repeatedly removes the last pattern-expression pair from the list to identify the first expression that causes a type error.

We used an incremental algorithm for match expressions rather than putting all the branches into a list (as we did for conditional expressions), because we want to check not only whether there is a type conflict but also which one is in conflict. The above algorithm identifies the first conflicting branch in the given match expression.

Using the extended type debugger, the error message of the example in Section 3.1 (match expression) becomes as follows:

```
Error:
The highlighted expression has type tree_t and
the previous expression has type 'a list,
but these should be the same type.
```
(highlight 2)

In addition to printing the type of the detected subprogram in the error message, the type debugger also prints other types (such as `'a list` in the example) to help programmers easily decide which one to fix.

5.1.3 Applications

Since the type debugger already shows which argument causes the type error in the error message, we did not change its basic behavior. Instead, we changed the error message to print the types of the function, all the arguments, and the required type for the detected expression.

For example, for the program in Section 1 where the programmer intends its type to be `int -> string`, we have the following error message:

level	1	2	3	4
partial application		✓	✓	✓
`if` without `else`, `unit`, side effects			✓	✓
Confusing operators (`==`, `!=`, `or`, `&`)				✓

Table 3: Language levels

```
fun x -> [(x + 1) ^ x]1
Error:
The first argument of this application causes a type error.  (highlight 1)
The types of the function, its arguments, and the required type for the first
argument are:
Function (^): string -> string -> string
First argument: int
Second argument: 'a
Required for the first argument: string
```

The type of the second argument is `'a`, because our type debugger is compositional: the second argument `x` does not impose any constraints on the type of `x`.

5.2 Introduction of language levels

One of the observations gleaned from the logs is that many students forgot to write some of the necessary arguments. Since OCaml is a higher-order language, passing fewer arguments does not necessarily mean a type error. However, it often results in a complicated error message later, which mentions higher-order types. To provide informative error messages in such cases, we extended the type debugger so that it would have language levels [2], following Racket (See Table 3).

Level 1. The Level 1 language is for beginners who do not know first-class functions. At this level, whenever an application is identified as the source of a type error, the type debugger checks the type of its arguments. If any of the arguments have higher-order types, it prints out the higher-order argument with its type and suggests that some arguments might be missing. This way, the type debugger can point out unintentional first-class function values. Such an error message can only be provided, because we assume that students do not use partial applications at this stage.

At language level 1, the error message for the example from Section 3.1 (application) becomes as follows:

```
(* f : int list -> int -> int list *)
(* g : int list -> int list *)
let test = [f (g lst) = [a; very; large; list; ...]] 1
Error:
The second argument of this application causes a type error.  (highlight 1)
Function (=): 'a -> 'a -> bool
First argument: int -> int list
Second argument: int list
```

Conditional		Match		Application	
Subprograms	%	Subprograms	%	Subprograms	%
Predicate	3.1	`pattern`	0	Argument	98.4
`then` not `unit`	78.1	`exp` and `pattern`	4.9	Difficult to explain	1.6
`then` and `else`	9.4	`expression`	80.4		
Difficult to explain	9.4	Difficult to explain	14.7		

Table 4: Simulation of 2012 course

Conditional			Match			Application		
Subprograms	Level	%	Subprograms	Level	%	Subprograms	Level	%
No `else`	1-2	1.5	`pattern`	1-3	4.9	Partial application	1-2	6.6
Predicate	1-3	0	`exp` and `pattern`	1-3	3.2	Argument	1-3	75.6
`then` not `unit`	3	4.5	`expression`	1-3	66.1	Non-function application	1-3	17.2
`then` and `else`	1-3	73.1	Difficult to explain	1-3	25.8	Difficult to explain	1-3	0.6
Difficult to explain	1-3	20.9						

Table 5: Preliminary analysis of 2014 course

```
The following arguments have the function type.
First argument: int -> int list
(some argument might be missing.)
```

At language levels higher than 1, the type debugger prints the types of the function and arguments. At language level 1, the type debugger additionally prints the types of the higher-order arguments.

Level 2. Level 2 language is for students who know first-class functions but do not use side effects. Side effects are introduced only near the end of the course. Until then, students are not allowed to use side effects and do not know the `unit` type.

We have modified the OCaml parser to prohibit expressions related to side effects and the `unit` type: `if` without `else`, `for`, `while`, assignments, and sequential execution. At language level (1 and) 2, we can explicitly point out that `else` is missing and suggest to add it:

```
fun x -> if true then x + 1   [1]
Error:
The else statement is missing.  (highlight 1)
```

Levels 3 and 4. We created one more level (level 3) before the full OCaml language (level 4). Language level 3 prohibits the use of confusing operators, `==` and `!=`. These pointer equality operators are not used in the course and are simply prohibited. Students are directed to use `=` and `<>`, respectively. Students are also directed to use `||` and `&&`, instead of deprecated operators, `or` and `&`, respectively.

6 Evaluation

We evaluated how much better the extended type debugger is in comparison with the previous one. First we simulated what would have happened if we used the extended type debugger for the 2012 course. We then did a preliminary analysis of the extended type debugger in the 2014 “Functional Programming” course. We examined the difficult cases that we encountered in Section 7.1.

6.1 Simulation of 2012 course

Assuming that students answer the same as in the logs, we manually simulated what would have happened if we had the extended type debugger for the 2012 logs (Table 4). The columns Conditional and Match in Table 4 summarize the results for the cases where the detected expressions are ill-typed conditional and match expressions, respectively. In both cases, we observe that the extended type debugger can provide more specific error messages most of the time. We describe the “Difficult to explain” cases in detail in Section 7.1.

For the cases where the detected expressions are ill-typed applications, we observed that in 98.4% of all cases, the type debugger could show enough information from which students should be able to understand the source of the type error. The remaining cases, 1.6% of all cases, are classified as difficult (see Section 7.1).

6.2 Preliminary analysis of 2014 course

We used the extended type debugger in the 2014 “Functional Programming” course. This year, all the user interactions, regardless of whether type errors occur, are logged, so that we can analyze students behavior more accurately. To ease the classification efforts, we numbered the error messages to indicate what the source of the type error is. Table 5 presents the initial results, showing the number of times the classified errors occurred.

Most of the type errors in conditional expressions come from type conflicts between `then` and `else` branches. There are only 1.5% of missing-`else` type error at language level 1. Similarly, the sources of type errors for match expressions are mostly from conflicting expressions. In both cases, the type debugger can provide specific error messages (except for “Difficult to explain” cases which are explained in the next section).

For applications, “Partial application” means that some of the arguments are higher order. Since partial applications are not used at level 1, this means that students did not pass enough arguments. The table shows that students often pass fewer arguments than necessary. The “Argument” means that all the arguments are first order. In this case, the type debugger displays all the type information of the function and its arguments.

Interestingly, not a few type application errors come from application of non-functions. Typically, this happens when students forget “;” in a list (e.g., `[2; 1; 4 5]` instead of `[2; 1; 4; 5]`) or when students forget an infix operator (e.g., `"hello" "world"` instead of `"hello" ^ "world"`). These cases are handled by the type debugger without any problems. We separated these cases in order to give a slightly better error messages, saying that the expression is parsed as an application but a function is missing.

Overall, compared with the original type debugger, the extended version detects smaller expressions as sources of type errors and gives more precise and informative error messages. We have found that more students are trying to use the type debugger compared with two years ago.

7 Limitations

Overall, our type debugger detects and expresses where the sources of the type errors are, but we nonetheless found that it has some limitations.

7.1 Difficult cases

As we classify the logs, we encounter difficult cases where all the subexpressions are well-typed and satisfy the necessary type constraints, but the whole expression does not type check. In all the cases, they have a variable with conflicting types. Even though all the subexpressions are well-typed independently, the same variable has one type in one subexpression and another type in other subexpressions. For example, consider the following expression:

```
fun p -> fun q -> if p and (q = 1) then p else
```

After identifying the conditional expression as the source of the type error, the extended type debugger checks whether subexpressions satisfy the type constraints as follows:

1. Annotate the predicate `p and (q = 1)` as `bool` and pass it to the compiler's type inferencer. It type checks with the environment {`p : bool, q : int`}.
2. Construct a list of branches `[p; q]` and pass it to the compiler's type inferencer. It type checks with the environment {`p : 'a, q : 'a`}.

Thus, each subexpression of the conditional satisfies the required type constraints independently. However, the two environments conflict with each other and are not unifiable.

It is not clear what informative error messages we can provide for such cases. Currently, our type debugger prints out that some variable is used at two different types. If we were to implement a dedicated type inferencer, we could say more specifically what had happened in this case. However, this is to the disadvantage of our type debugger, where the compiler's type inferencer is reused, thereby making it easy to build a type debugger that is consistent with the compiler's type inferencer. It is not clear either if a dedicated type inferencer could actually give a better error message. In the future, we will investigate how serious this case is for novice programmers and consider what we can do about it.

7.2 Type variables

In compositional type inference, since a variable does not impose any constraints on its type, the type of a variable always exists: a type variable. Although the compositional type inference helps to identify where the type of a variable is first forced to a specific type, showing type variables in the error message can sometimes confuse students. For example, the following program is taken from the 2012 logs:

```
let rec search tree name = match tree with
        Empty -> Empty
      | Node (t1, st, n, t2) -> if (st = name) then n
                                else [search search t2 name]1
Error:
The first argument of this application causes a type
error. (highlight1)
The types of the function, its arguments, and the required type
for the first argument are:
```

```
Function: 'a
First argument: 'b
Second argument: 'c
Third argument: 'd
Required for the first argument: 'e
```

The type debugger reports (without asking any questions) that the last application is the source of the type error. It does not ask any questions, because the application does not type check but all its subexpressions trivially type check. (Remember a variable always type checks.) As a result, the extended type debugger lists all the arguments whose types are all type variables.

One might think that we can at least say that the type of function and the type of the first argument are the same type variable. To do so, however, one is probably required to implement a dedicated type inferencer. Even if we could implement it, it is not at all clear how to explain why the type error occurs.

7.3 Users' input

Our type debugger requires users' input. This can be both an advantage and a disadvantage. Without asking questions, it is impossible in general to locate a single source of a type error that reflects the users' intentions. By asking questions, our type debugger exactly locates the source of a type error. On the other hand, it means that the result depends on the users' answer. If a student inputs a wrong answer for some reason, e.g., as a result of misunderstanding the questions, or a lack of knowledge on types, the type debugger would identify a different place as the source of the type error. From our experience so far, we feel that questions raised by the type debugger make students think about types of expressions more seriously, which is nice from the pedagogic point of view. However, we need more experience to draw a definite conclusion.

8 Related work

To produce understandable and appropriate error messages, Heeren, Hage, and Swierstra [6] divided type inference in the Helium compiler for Haskell into two phases: generation of type constraints and solving of type constraints. When unsolvable type constraints are found during the constraint satisfaction phase, they are output as type error messages. By controlling which type constraint to remove from the unsolvable ones, we gain control over which expression to blame. By registering a specific error message to each type constraint, we can not only tailor type error messages [7], but also show possible fixes for typical error cases, such as proposing similar 'sibling' operators. As such, Helium has been successfully used in a classroom setting with many expression-specific error messages [5].

We share with Helium's authors the goal of producing better error messages. To ascertain the programmers' intentions, we employ an interactive approach where the debugger asks questions on the types of expressions. Once the source of a type error is identified, the technique used in Helium will be valuable to enhance error messages.

Interactive type debugging was proposed by Chitil [1] for a small subset of Haskell. This was followed by Tsushima and Asai [16] for full OCaml. A similar approach was taken by Stuckey, Sulzmann, and Wazny [15], who implemented an interactive type debugger for Haskell (including various advanced features). As in Helium, they produce type constraints and find the minimal conflicting set to narrow down the possible cause of type errors. By adding type constraints interactively, the programmer can express his intention and be led to the sources of the type errors.

A conflicting set of type constraints is called a type error slice [4]. Among the type error slices, the minimal one is useful for type debugging because it contains the smallest set of type constraints that lead to a type error. The minimal type error slice can be automatically obtained without any inputs on the programmer's intention. Rahli, Wells, and Kamareddine implemented a type error slicer for the full set of SML [11].

Writing a dedicated type error slicer is not an easy task. Schilling [12] designed a type error slicer without implementing a dedicated type inferencer but by reusing the compiler's type inferencer. Tsushima and Asai followed this approach [17]. Lerner, Grossman, and Chambers used the compiler's type inferencer to propose a possible fix to a type error. They enumerated possible changes to an ill-typed program and check whether the modified programs type check using compiler's type inferencer as a black box. If they type checked, they are shown as possible fixes of the type error.

The compositional type inference used in our type debugger is based on the one proposed by Chitil [1]. Compositional type inference can identify where the type of a variable is first fixed locally without being affected by the surrounding expressions. A similar approach is used in the $\mathscr{U}_{AE}$ system by Yang [18], which can report which parts of a program have type conflicts. Note that the $\mathscr{U}_{AE}$ system is in accord with most of the manifesto for good error messages [19].

There are also many studies and user tests about the reactions of novice programmers while they are programming. Marceau et al. reported on novice programmers using Racket [8, 9]. They analyzed the effectiveness of each error message of the Racket language. We basically followed their advice in designing the error messages of our type debugger. There is a report [10] that classified and discussed the efficiency of debugging methods for Java programs (e.g., inserting printf, use of JavaDoc or debuggers, and narrowing erroneous programs by commenting out). While they focused on the methods of debugging, our user tests focused more on the reactions of programmers. Spohrer and Soloway analyzed bugs in Pascal code written by novice programmers [14]. They classified bugs not only by expressions but also by the users' intention ("plan"). They categorized bugs into "correct plan but wrong implementation" and "wrong plan and implementation". This paper also focused on the intentions of users in order to detect the sources of type errors.

The idea of pedagogical language levels is attributed to Felleisen, Findler, Flatt, and Krishnamurthi [2]. They proposed an alternative role for functional programming for novices. The idea of dividing a language into levels is widely used in the Racket language. It has many useful language levels from beginners to experts. The Racket language levels are used not only for providing better error messages but also for avoiding unnecessary advanced syntax, among others. We imported this idea to improve the error messages of the type debugger. The Helium compiler employs simple language levels via a flag. When the flag is off, overloading is turned off and error messages no longer mention type classes.

9 Conclusion and future work

In this paper, we analyzed the logs of our interactive type debugger from the 2012 functional language course in our university. Although the type debugger worked well for some cases, the analysis showed that expression-specific and more informative error messages are desired. According to these observations, we extended the type debugger to provide expression-specific error messages for conditional and match We evaluated the resulting type debugger with logs from 2012 and showed a preliminary analysis of the logs from the 2014 course. In both cases, we showed that the type debugger provides more informative error messages in most of the cases. We also showed difficult cases where a variable was used in conflicting types.

Although a thorough evaluation of the type debugger requires a full analysis of the logs from the 2014 course, we have the impression that the type debugger deals with most of the common type errors properly.

We plan to further pursue the following directions in the future. First, we want to analyze the new logs from the 2014 course and see if there are cases the current type debugger cannot handle well. We already found some cases where expression-specific handling is required, such as `try with` blocks. Supporting them is essential to making the type debugger robust. On a related note, an automatic log analysis could be an interesting topic. The amount of logs is becoming large and it is simply unrealistic to manually analyze them.

Once the type debugger covers most of the common errors, we want to build a taxonomy of common type errors of novice programmers. We could then create a document describing typical errors, which novices can study. In particular, we found type errors that can be identified by the type debugger but more information in the error message would be useful. For example, not a few students write only an exception (e.g., `Not_found`) without raising it. Another example is to forget writing a dereference (`!`) before a reference. In such cases, the type debugger could show typical related mistakes.

We also plan to use the type debugger in the upper level course to see if it is useful for medium-level programmers. In that case, the programs to be debugged are large and the number of questions will increase. To reduce the number of questions, we could introduce type-error slices. The type debugger is not only for novice programmers: it is also for experts to use.

10 Acknowledgement

We appreciate the valuable feedback and constructive comments by the reviewers.

References

[1] O. Chitil. (2001): Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In: *ICFP '01 Proceedings of 6th ACM SIGPLAN International Conference on Functional Programming*, pp. 193–204, doi:10.1145/507656.507659. Available at http://dl.acm.org/citation.cfm?id=507659.

[2] M. Felleisen, R.B. Findler, M. Flatt & S. Krishnamurthi (2004): The Structure and Interpretation of the Computer Science Curriculum. In: *Journal of Functional Programming*, pp. 365–378, doi:10.1017/S0956796804005076. Available at http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=227683&fileId=S0956796804005076.

[3] M. Felleisen, R.B. Findler, M. Flatt & S. Krishnamurthi (2014): How to Design Programs, 2nd edition. Cambridge: MIT Press.

[4] C. Haack & J.B. Wells (2004): Type Error Slicing in Implicitly Typed Higher-Order Languages. In: *Science of Computer Programming, the 12th European symposium on programming (ESOP'03)*, 50, pp. 189–224, doi:10.1016/j.scico.2004.01.004.

[5] J. Hage & B. Heeren (2007): Heuristics for Type Error Discovery and Recovery. In: *Implementation and Application of Functional Languages (IFL'06)*, 4449, pp. 199–216, doi:10.1007/978-3-540-74130-5_12.

[6] B. Heeren, J. Hage & S.D. Swierstra (2003): Constraint Based Type Inferencing in Helium. In: *Workshop Proceedings of Immediate Applications of Constraint Programming*, pp. 59–80.

[7] B. Heeren, J. Hage & S.D. Swierstra (2003): Scripting the Type Inference Process. In: *the 8th ACM SIGPLAN international conference on Functional programming (ICFP'03)*, pp. 3–13, doi:10.1145/944705.944707. Available at http://dl.acm.org/citation.cfm?doid=944705.944707.

[8] G. Marceau, K. Fisler & S. Krishnamurthi (2011): Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In: *SIGCSE '11 Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 499–504, doi:10.1145/1953163.1953308. Available at http://dl.acm.org/citation.cfm?id=1953163.1953308.

[9] G. Marceau, K. Fisler & S. Krishnamurthi (2011): Mind Your Language: On Novices' Interactions with Error Messages. In: *ONWARD '11 Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 3–18, doi:10.1145/2048237.2048241. Available at http://dl.acm.org/citation.cfm?id=2048241.

[10] L. Murphy & et al. (2008): Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies. In: *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pp. 163–167, doi:10.1145/1352135.1352191. Available at http://dl.acm.org/citation.cfm?id=1352191.

[11] V. Rahli, J.B. Wells & F. Kamareddine (2010): A constraint system for a SML type error slicer. Technical Report, Technical Report HW-MACS-TR-0079, Heriot-Watt university.

[12] T. Schilling (2012): Constraint Free Type Error Slicing. In: *the 12th international conference on Trends in Functional Programming (TFP' 11)*, pp. 1–16, doi:10.1007/978-3-642-32037-8_1. Available at http://link.springer.com/chapter/10.1007%2F978-3-642-32037-8_1.

[13] E.Y. Shapiro (1983): Algorithmic Program Debugging. Ph.D. thesis, Yale. Available at http://dl.acm.org/citation.cfm?id=538679.

[14] J.C. Spohrer & E. Soloway (1986): Novice Mistakes: Are The Folk Wisdoms Correct? *Communications of the ACM* 29, pp. 624–632, doi:10.1145/6138.6145. Available at http://dl.acm.org/citation.cfm?id=6145.

[15] P.J. Stuckey, M. Sulzmann & J. Wazny (2003): Interactive Type Debugging in Haskell. In: *The 2003 ACM SIGPLAN workshop on Haskell*, pp. 72–83, doi:10.1145/871895.871903. Available at http://dl.acm.org/citation.cfm?doid=871895.871903.

[16] K. Tsushima & K. Asai (2012): An Embedded Type Debugger. In: *Implementation and Application of Functional Languages (IFL' 12)*, 8241, pp. 190–206, doi:10.1007/978-3-642-41582-1_12. Available at http://link.springer.com/chapter/10.1007/978-3-642-41582-1_12.

[17] K. Tsushima & K. Asai (2013): A Weighted Type Error Slicer. In: *Implementation and Application of Functional Languages, Lecture Notes in Computer Science 8241 (IFL' 13)*, pp. 190–206.

[18] J. Yang (2010): Explaining Type Errors by Finding the Source of a Type Conflict. In: *Trends in Functional Programming*, pp. 58–66. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.8146.

[19] J. Yang, G. Michaelson, P. Trinder, & J.B. Wells (2000): Improved Type Error Reporting. In: *[Draft] Proc. 12th Int'l Workshop on Implementation of Functional Languages, Aachen, Germany*, pp. 4–7. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.6846.