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 [5, 8, 7] interactively detects the expression which causes a type error. It asks users whether they intend the types of identifiers to be those that the compiler inferred. However, it seems that novice programmers using type debugger often get in trouble with conceiving the solutions of the type errors. In this paper, we analyze the user testing of type debugger and extend it. Furthermore, we also introduce language levels to OCaml by editing the parser. At last, this paper shows some example of error-logs which we think difficult to explain the sources of the type errors. The target of the user testing is 40 novice students belong to the department of information science, Ochanomizu University.

1 Introduction

Strongly-typed languages, such as OCaml or Haskell, allow programmers to write exectable programs by matching the types in programs. For example, the OCaml compiler prints an error message when a programmer defines a function which calculates x-th power of `(x + 1)` like below:

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

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

This error messaage points out that the types of "`^`" (`string -> string -> string`) conflicts with "`(x + 1)`" (`int`). The programmer can resolve this type error and make an exectable program by changing the code in either way:

1. Apply "`int -> int -> int`" function which calculates the power instead of "`^`".
2. Change the type of "`(x + 1)`" and "`x`" to "`string`" by applying "`string_of_int`".

In this case, since the programmer wants "the power of an integer (which is also an integer)", he/she could choose the first way to fix the program.

Tsushima and Asai [8, 7] proposed and implemented a type debugger which reuses the type inferencer of the compiler. Using the most general type tree [5] for type inference and Algorithmic Program Debugging [10], the type debugger can detect the source of the type errors. In this paper, we aimed to extend the type debugger in order to make it novice-friendly by analyzing the user testing of the debugger targeted beginner programmers who are not used to write strongly-typed languages.

The paper is structured as follows. In section 2, we briefly show how our type debugger works. In section 3, we analyze the user testing of the debugger and discuss the result in section 4. In section 5, we briefly introduce some points to design error messages for novice programmers by Marceau et al. [3]. We extend the debugger and OCaml parser in section 6. Related works is discussed in section 7, and in section 8, we conclude the paper.

DRAFT April 21, 2014

2 Type debugger

We show how the type debugger works in this section. The debugger constructs the most general type tree [5], and uses Algorithmic Program Debugging [10] to detect type errors.

2.1 Most General Type Tree

The most general type tree (MGTT) which the type debugger uses is the compositional type tree [5] devised by Chitil. The MGTT of the previous program is as shown below (where $\tau^+ = \texttt{int} \to \texttt{int} \to \texttt{int}$ and $\tau^\wedge = \texttt{string} \to \texttt{string} \to \texttt{string}$).

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

In the standard typed tree, type unification makes $\{\texttt{x : int}\}$ in the box B be propagated to the box A. Suppose the programmer intends $\{\texttt{x : string}\}$, in other words, he/she needs to fix it to `fun x -> (string_of_int (x + 1)) ^ (string_of_int x)`. The debugger could not find out when `x` was forced to have type `int`. But when we see the each expression independently, `x` in the box A could be just type variable `a` instead of type `int`. Since MGTT composes each type of expressions from their sub-expressions, in this case, it is not until the node with box B is composed from three nodes with box A that the type of `x` turns into `int`. Using MGTT, the debugger can find out that the box B is the source of this type error.

2.2 Algorithmic Programming Debugging

The type debugger detects the sources of type errors by walking MGTT with Algorithmic Programming Debugging (APD). APD is used to detect errors in a tree structure, and originally devised to find errors in Prolog programs by Shapiro [10]. The algorithm starts from an error node:

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

Judging whether an error occurs is usually done by an input from users. In our type debugger, APD uses whether the types of environments and expressions inferred by compiler match to users' intention or not.

2.3 Detecting type errors

The type debugger detects the sources of type errors in two steps:

1. Find a node (expression) which has a type error, but all of its children are well-typed.
2. Find the expression whose children have well-intended types, but the expression is not well-intended.

For the first step, the debugger uses the results of compilers' type inference for APD. Then the debugger asks users whether the types inferred from compilers match their intention at the second step.

expression	%
Appliation	27.2
Match expression	12.1
Constructor	10.2
If expression	4.1
Recursive function	3.43
Environment	17.0
Syntax misunderstood	16.3
Failed to classify	6.0
Undefined variable	3.8

Table 1: Classification by expression

3 Analysis

We analyzed the type-error logs of our type debugger. The target of this user testing is 40 CS-major students in Ochanomizu University who registered the "Functional Programming" course in 2012. Although they have one year experience of writing C programs, it was their first experience to write strongly-typed languages, OCaml in this class. We analyzed the type-error logs in two ways:

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

3.1 Analysis by each expression

Table 1 shows the breakdown of expressions detected as sources of type errors by the type debugger. In this section, we see some typical type errors from the error log.

Application 27.2% of the source of type errors were located in application, which gained the highest percentage of all expressions. The type debugger finds out which argument causes the type error by passing the increasing number of arguments to the function until type error occurs. A typical type error of using application is as below. Boxes in programs show high-lights.

```
fun x -> (x + 1) ^ x
```[1]

```
Error:
The first argument of this application causes a type error. (high-light 1)
```

In this case, "`this application`" refers to the "`^`" operation and "`the first argument`" refers to "`(x + 1)`". However, since the "`^`" operation is used in infix notation, the most students seemed to get in trouble finding out which expression or identifier was "`this application`".

In other logs, some students passed less arguments than the function expected. Here is a simple example we often saw:

```
(* f : int list -> int -> int list *)
```

```
(* g : int list -> int list *)
let test = g (f lst) = [a; very; large; list; ...]
```
[1]

```
Error:
The second argument of this application causes a type error. (high-light 1)
```

This error message suggests that "`[a; very; large; list; ...]`" is the source of the type error, but acutually it is not. What causes this type error is that the student only passed one argument to function `f`, which needs two arguments, and the type of the "`=`" operator became `(int -> int list) -> (int -> int list) -> bool`. Since the error message mentions only the "`second argument`", students only checked the large list and rarely found out they forgot an `int` argument for `(f lst)`. Even if the students could understand "`this application`" refers to the '`=`" operator, it seems to be difficult for them to fix the appropriate expression by this error message.

match expression As the course goes forward, programs get more and more complicated. One of the large expressions students write is match expressions. Here is an example that a student changed for more than 10 times to fix the type error:

```
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 =
```
[3]

```
  match station with
   [] -> []
   |{start = st; destination = dest; distance = dist;} :: rest ->
```
[1]

[2]

```
     match station_tree with
       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:
Something in this match expression causes the type error. (high-light 3)
```

While the source of this type error is high-light 1, where the student returns a list instead of a tree, the debugger showed the whole match expression in high-light 3. Even though the source of the type error was simple, high-ligth 3 was too large and complicated to find it out.

If expression Most of the type errors located in if expression did not have `else` statement. In this case, if expression should have type `unit`, but students wrote expressions of some other types. After the user testing, one of the students told us she did not understand why if expression must have type `unit`. It could mean that she did not know if expression must be of type `unit` when it does not have `else` statement. Also, since the debugger high-lights whole the if expression and prints the same messages

	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
If expression	0	50.0	0	50.0
Recursive function	7.7	7.7	23.1	61.5

Table 2: Analysis by reaction of users (%)

whenever if expression (or match expression) is located as the source of type errors, students could rarely detect the source of type errors.

3.2 Analysis by reaction of users

We classified how students changed the programs after they read the error messages of the type debugger in four ways below. We followed [3] for this classification.

1. An effective change to fix the type errors.
2. The change to fix appropriate expression, but ineffective.
3. The change to fix an expression unrelated to the type errors.
4. No change, such as inserting indentation or spaces.

Table 2 shows the classification of the way in which students changed the programs. We found that very few programs could be corrected effectively. For example, students could fix almost 40% of type errors located in application, but for the other cases, they could not find out efficient ways to fix type errors. Furthermore, more than 60% of type errors located in recursive function were left unchanged.

We analyzed application, match expression, and if expression below.

Application When an application is detected as the source of a type error, the type debugger prints the argument which causes the type error. So the students often changed that argument or swap the arguments. However, they seemed to be in torment when the infix notation was used, or the argument was unintentionally higher-order (such as we saw in section 3.1).

Match expression Students tend to change a wrong expression when the high-light covers 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 find out where to fix. This reaction contributes to 38%, relativey high percentage, of effective change.

If expression As we saw in section 3.1, students tend to get in trouble with the case when they did not write `else` statement. In this case, they often changed the `then` statement into something else, and rarely found out they could just add `else` statement.

4 Discussion

From the analysis in section 3, we disucuss our strategy to extend the type debugger for novice programmers.

1. Smaller high-lights:
   the debugger needs to detect and high-light as smaller expression as possible instead of whole the expression.
2. Novice-friendly error messages:
   the debugger needs to give more information not only about the very source of the type error but also related expressions or environments.
3. Language levels:
   We found that at the beginning of the "Functional Programming" course, students do not need to use higher-order functions, type `unit`, and side effects. This means that we can divide the OCaml language into several levels from a beginner language to an expert language.

Before we extend the type debugger, we discuss relation between "novice-friendly" and "educational". Here is a program which was found in the type-error logs (the student seemed to be on her way to change the "`fold`" for a list to the "`fold`" for a tree):

```
let rec fold f init lst = match lst with
   [] -> init
   | first :: rest -> f first (fold f init rest)

let length lst = fold (fun l _ _ r -> l + r + 1) 0 lst
```
[1]

```
Error:
The second argument of this application causes the type error. (high-light 1)
```

At first, the type debugger passes the first argument (`(fun l _ _ r -> l + r + 1)`) to the function `fold`. Since the function `fold` is quite generic, the type error does not occur even if a strange function such as `(fun l _ _ r -> l + r + 1)` was passed. The type of `fold (fun l _ _ r -> l + r + 1)` is `('a -> int -> int) -> int list -> 'a -> int -> int` and it requires `('a -> int -> int)` for the next argument (`0`), which has type `int`. This is why the error message suggests to fix the second argument while the source of the type error is the first argument.

Watching carefully into the definition of `fold`, `f` is applied to two arguments. Therefore, the debugger might be able to guess the first argument of `fold` is the source of the type error, because it receives four arguments. Rather than persuing this approach, however, we choose more educational approach: by showing all the types of arguments, we force students to think what went wrong. It has also an advantage to avoid guessing uncertain user intention that leads to incorrect error messages.

In the next section, we introduce the report of Marceau [2, 3] to design suitable error messages.

5 Error messages for novice programmers

Marceau et al. analyze and discuss the error messages for novice programmers using Racket [3]. In particular, these are what we found remarkable:

- Even English-speaking students do not understand the meaning of error messages in English, although they know each vocabulary in the messages.
- Some students do not know the terminology of functional programming.
- Some students do not find out what demonstratives (such as "This expression") refer to.

Then Marceau et al. note two points to design error messages in view of education:

- Error messages should not point out the solution for debugging. The solutions suggested can never be really enough for all the cases even if the error was simple.
- Error messages must not make programmers debug in a wrong way. For example, high-lights should not allow the programmers to think it enough to fix only inside the high-lights.

Since our target is also computer-science students, we design and extend the type debugger and parser almost in line with these two points. Moreover, we add another point:

- Error messages of type debugger should help novice programmers understand the type system of OCaml.

Here is an example code:

```
...  if p < q then p + 1 else "q"
```

If the programmer did not know types of `then` statement and `else` statement must not conflict with each other in OCaml, he/she may not be able to debug this program only by the message "if expression causes the type error". Therefore, we changed the error message to print those below:

1. `then` statement and `else` statement conflict with each other.
2. The types of `then` and `else` statements.
3. `then` and `else` statements need to be the same type in if expression.

What we think most important for error messages of the type debugger is to give programmers opportunity to learn the type system.

6 Extension

In this section, we show how we extend the type debugger and OCaml parser.

6.1 Type debugger

As we saw in section 2, the most general type tree (MGTT) was necessary to detect the expression causing a type error. In order to detect the subprogram, we need to add types to each subprogram in MGTT. The type debugger checks whether each subprograms has the type along with constraints of OCaml type system. For example, suppose we are writing if expression. Then the predicate needs to have type `bool`, and `then` statement and `else` statement need to be the same type. If the type of a subprogram is against the type system, the type debugger detects that subprogram to be the source of the type error.

We extended the type debugger to be able to detect a subprograms of if expression and match expression, and changed error messages of application.

6.1.1 if expression

In OCaml, the constrains of if expression are as below:

1. The predicate must be of type `bool`.
2. If the expression does not have `else` statement, `then` statement must be of type `unit`.
3. If the expression has `else` statement, `then` and `else` statements must be the same type.

The type debugger checks these three constrains in the following algorithm:

1. The debugger adds the `bool` annotation to the predicate statement, and infer its type using the type inferencer of OCaml compiler. If the statement results in a the type error, the debugger detects the predicate to be the source of the type error. If not, go to the next step.
2. If the expression does not have `else` statement, the debugger checks whether the `then` statement has type `unit` as the same way as in the first step.
3. The debugger creates a new list containing `then` statement and `else` statement, so that the debugger can annotate the same type for those statements. The debugger gives the list to the type inferencer of the compiler to check if it leads to a type error.

Show three improved examples from the type-error logs.
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.   (high-light 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.
(or you might forget else statement) (high-light 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. (high-light 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 high-lights, where only detected subprograms are colored.

6.1.2 match expression

The match expression in OCaml consists of `(exp, (pattern, expression) list)` where `exp` is the expression to be matched, `pattern` the left-hand side of the arrows, and `expression` the right-hand side of the arrows. The constraints of match expressions in OCaml are the following:

1. Every `pattern`s has the same type.
2. `exp` and `pattern`s have the same type.
3. Every `expression`s has the same type.

In order to check these rules, we extended the debugger with the following algorithm. Basically, it searches the source of the type error by removing `(pattern, expression)` from the list one at a time until the type error is gone:

1. Give `(pattern, dummy-expression) list` to the type inferencer of the compiler where all the `dummy-expression`s have the same type, so that the debugger can judge whether the source of the type error is in `pattern`s or not. If this dummy-match expression did not cause a type error, go to step 5.
2. Remove the last `(pattern, expression)` from the list and check if a type error occurs.
3. Repeat the step 2. while the type error occurs.
4. If the type error does not occur, judge the `pattern` that has been just removed as the source of the type error. If the type error occurs after all the elements are removed, detect the source of the type error as caused by the conflict between the types of `exp` and `pattern`s.
5. Give `(pattern, expression) list` to the type inferencer of the compiler and repeat the step 2. while the type error occurs.
6. When the type error does not occur, judge the `expression` that has been just removed as the source of the type error.

Using the extended type debugger, the error message of the example in section 3.1 (match expression) becomes the following:

```
Error:
The high-lighted expression has type tree_t and previous expression has type 'a list,
but these should be the same type.
```
(high-light 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.

6.1.3 Application

Since the previous type debugger already showed which argument causes the type error in the error messages, we did not changed the basic algorithm. Instead, we changed the error messages to print the types of the function, all the arguments, and required types for the detected expression.
Here we have the example from section 1:

`fun x ->` `(x + 1) ^ x`[1]

```
Error:
The first argument of this application causes the type error. (high-light 1)
The types of the function, arguments, and required type for the first argument are
following:
Function (^): string -> string -> string
First argument: int
Second argument: 'a
Required for the first argument: string
```

The example from section 4:

```
let rec fold f init lst = match lst with
   [] -> init
   | first :: rest -> f first (fold f init rest)

let length lst = fold (fun l _ _ r -> l + r + 1) 0 lst
```
[1]

```
Error:
The second argument of this application causes the type error.  (high-light 1)
The types of the function, arguments, and required type for the second argument are
following:
Function (fold): ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
First argument: int -> 'c -> 'd -> int -> int
Second argument: int
Third argument: 'e
Required for the second argument: 'f -> int -> int
```

It may be still difficult for novice programmers to find out the solution to fix the type error of the second example (from section 4), but we expect them to be used to checking types and trying to match them.

6.2 Introduction of language levels

levels	expressions left out from the language
1	if expression without `else` statement higher-order function side effects confusing operators
2	if expression without `else` statement side effect confusing operators
3	confusing operators
4	

Table 3: Language levels

Since our "Functional Programming" course does not use higher-order function, type `unit`, and side effect at the beginning, we decided to introduce several language levels. We could then detect the sources of the type errors in more detail for unintentional higher-order function and if expression without `else` statement. We prohibit side effects because we teach functional programming and students do not use side effects at the beginning. Moreover, we prohibited to use some operators which make novice programmers confused. The features of each language level are shown in Table 3. The level 1 is the very beginning language and 4 is the fullset of OCaml.

All of the prohibition other than high-order functions basically are done by extending the parser. Prohibiting higher-order functions, on the other hand, needs extension of the type debugger.

In detail, we prohibited:

- higher-order functions at level 2 only when it causes type errors. When a type error occurs in the application, the debugger checks if all the functions are applied to enough arguments.
- for expression, while expression, assignment, and sequencial execution for side effects.
- "`or`", "`&`", "`==`", and "`!=`" for confusing operators.
  (lead users to "`||`", "`&&`", "`=`", and "`<>`").

Here are some examples that are improved:

At level 1 (example from section 3.1):

```
(* f : int list -> int -> int list *)
(* g : int list -> int list *)
let test = g (f lst) = [a; very; large; list; ...]
```
[1]

```
Error:
The second argument of this application causes a type error. (high-light 1)
Function (=): 'a -> 'a -> bool
Second argument: int

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

At the expert level, the type debugger printed the types of all the arguments. At the beginning student level, on the other hand, the debugger prints the type of the detected argument and the types of arguments that have function type. The debugger can suggest some arguments might be missing in the argments having function type because students taking the "Functional Programming" course do not use higher-order functions at this level.

At level 2:

```
if true then ()
```
[1]

```
Error:
you might forget else statement. (high-light 1)
```

Since students do not use type `unit` at this level in the course, the type debugger can suggest students add `else` statement.

subprograms	%
Predicate	3.1
`then` is not type `unit`	78.1
`then` and `else`	9.4
Difficult to explain	9.4

Table 4: if expression

subprograms	%
`pattern`	0
`exp` and `pattern`	4.9
`expression`	80.4
Difficult fo explain	14.7

Table 5: match expression

Argument	98.4
Difficult to explain	1.6

Table 6: application (%)

7 Discussion for extended type debugger

We tested the extended type debugger using the same error logs assuming that users interact with the type debugger the same way as before. Tables 4 to 6 show the parcentages of subprograms detected by the extended type debugger. Compare to the previous debugger, majority of type errors are properly classified leading to more detailed error message. However, there are still logs which are diffibult to explain.

In this section, we analyze and discuss the examples which are difficult for the debugger to explain. Most of those difficult logs have type variables in the error messages. In particular, in if expression and match expression, most logs which are hard to explain have conflicts in the environment. In other words, each subprograms are well-typed (does not have type errors), and also, they are well-intended, but the variables have different types in each subprogram. Suppose we write the following code:

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

This is how the type debugger works:

1. Annotate the predicate `p and (q = 1)` with type `bool` and give it to the compiler.
   The environment {`p : bool, q : int`}
2. Put `then` statement and `else` statement into the same list and give it to the compiler.
   The environment should be {`p : 'a, p : 'a`}

While both 1. and 2. do not cause type errors, the two environments cannot be composed into a single environment. Of course, we can detect which identifier is the source of the type error by implementing a dedicated type inferencer. However, this would be against the advantage of our type debugger reusing the type inferencer of the OCaml compiler. It would cost much to implement.

8 Related Work

There are many researches and user testing about the reactions of novice programmers during programming. Marceau et al. reported the user testing of novice programmers using Racket [2, 3]. In this paper, Marceau et al. analyzed and discussed the effectiveness of each error message of the Racket language. We introduced the details in the section 4. There is a report that classified and discussed the efficiency of the means of debugging while writing Java (e.g. print debugging, use JavaDoc or debuggers, comment-out debug) [4]. Our way to debug is the Type-debgger, and while they focused on the means of debugging, our user testing focused on the reaction of the programmers. James and Elliot analyzed the bugs in Pascal code written by novice programmers [6].They classified bugs not only by expressions but also by users' intention ("plan") to debug. They categorized the bugs into "correct plan but wrong

implementation" and " wrong plan and implementation". This paper also focuced on the intention of users to detect the sources of the type errors.

For dividing a language into levels, Racket[1] already has useful language levels from beginner to expert. While our approach detects higher-order functions only when a type error occurs, Racket in the beginning student language mode always prohibits higher-order functions.

9 Future Work

We consider to work on the following problems.

9.1 Type errors caused from misunderstanding syntax

"Syntax misunderstood" in Table 1 is specially separated from "Environment", because their percentage is high and these are typical errors for novice programmers. The following list shows what kind of type errors we encountered.

- Intended to write a list of integers, but actually wrote a list of tuples.
  (e.g. `[1, 2]` instead of `[1; 2]`)
- Intended to write a `tuple` but actually wrote a `float`.
  (e.g. `(1.2)` instead of `(1,2)`)
- After having defined
  `type tree_t = Empty | Node of tree_t * 'a * tree_t`,
  some students did not write the constructor `Node` to make a tree.
  (e.g. in a recursive function `double_tree`, which doubles all the elements in `tree_t`,
  `(double_tree left, 2 * n, double_tree right)` instead of
  `Node (double_tree left, 2 * n, double_tree right)`)
- Did not write `!` when referring to the element of type `ref`.
  (e.g. `counter := counter + 1` instead of `counter := !counter + 1`)
- Did not write `raise` when rasing exception.
  (e.g. `Not_found` instead of `raise Not_found`)

Since students were not used to writing OCaml, they often got in trouble with finding out those were just syntax problems. We consider to introduce the method of Benjamin [1]. It suggests alternative expressions for solution if it does not cause type errors when adapted instead of the similar expression which incurs a type error.

9.2 Introduction of type-error slice

Currently, we are undergoing the user testing of extended type debugger and parser in the same course. After analyzing this user testing, we would like to test it with larger programs. For larger programs, in order to make debugger's questions short, we consider to introduce type-error slice [9]. The type debugger could be not only for novice programmers but also for experts to use.

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

References

[1] Benjamin Lerner, Dan Grossman & Craig Chambers (2006): SEMINAL: Searching For ML Type-Error Messages. In: *ML '06 Proceedings of the 2006 workshop on ML*, pp. 63–73.

[2] Guillaume Marceau, Kathi Fisler & Shriram 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.

[3] Guillaume Marceau, Kathi Fisler & Shriram 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.

[4] Laurie 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.

[5] Chitil O. (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.

[6] James C. Spohrer & Elliot Soloway (1986): Novice Mistakes: Are The Folk Wisdoms Correct? *Communications of the ACM* 29, pp. 624–632.

[7] K. Tsushima & K. Asai (2012): An Embedded Type Debugger. In: *Implementation and Application of Functional Languages, Lecture Notes in Computer Science 8241 (IFL' 12)*, pp. 190–206.

[8] K. Tsushima & K. Asai (2012): A Type Debugger Using Type Inferencer of Compilers (in Japanese). In: *The 13th JSSST Workshop on Programming and Programming Languages*. 15 pages.

[9] K. Tsushima & K. Asai (2013): Generating A Type Error Slice By Type Inferencer of Compilers (in Japanese). In: *The 14th JSSST Workshop on Programming and Programming Languages*. 15 pages.

[10] Shapiro E. Y. (1983): Algorithmic Program Debugging. In: *MIT Press*.