Report on an OCaml Type Debugger

Kanae Tsushima
Ochanomizu University
tsushima.kanae@is.ocha.ac.jp

Kenichi Asai
Ochanomizu University
asai@is.ocha.ac.jp

1 Introduction

Strongly typed languages enable us to reduce runtime errors by eliminating type errors at compile time. However, it is not always easy to find the source of type errors from error messages given by a compiler.

For example, Figure 1 shows a small program with an error written in OCaml. The function `f` is supposed to receive a list of numbers and return a list of the positive numbers among them. Because the programmer wrote "`n`" in the predicate part of the if-expression instead of "`n > 0`", however, the function `f` is typed `bool list` → `bool list`, resulting in an error at `f [1; -1; 3]`:

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

Note that the intended source of the error is in function `f` and is different from the error message. Furthermore, the type of `f` at "`f [1; -1; 3]`" is constrained to `bool list` → `bool list` although intended type of `f` is `int list` → `int list`.

```
(* f : int list -> int list *)
let rec f lst = match lst with
  | [] -> []
  | n :: tl -> if n then n :: f tl else f tl
in f [1; -1; 3]
```

Figure 1. Example program

$$\dfrac{\{n{:}a\}\vdash n{:}a \qquad \dfrac{\dfrac{\{\}\vdash (::){:}pl \quad \{n{:}b\}\vdash n{:}b}{\{n{:}b\} \vdash (n\ ::){:}b\ list{\to}b\ list} \qquad \dfrac{\{f{:}c\}\vdash f{:}c \quad \{tl{:}d\}\vdash tl{:}d}{\{f{:}d{\to}e,\ tl{:}d\} \vdash f\ tl{:}e}}{\{n{:}b,\ tl{:}c,\ f{:}c{\to}b\ list\} \vdash n{::}(f\ tl){:}b\ list}}{\{n{:}bool,\ tl{:}c,\ f{:}c{\to}bool\ list\}\vdash \text{if-exp}{:}bool\ list}$$

pl = x → x list → x list (from polymorphic environment)
if-exp = if n then n :: (f tl) else (f tl)
Note: we omit the else-part

Figure 2. Compositional type inference

$$\dfrac{\Gamma\vdash n{:}bool \qquad \dfrac{\dfrac{\Gamma\vdash (::){:}pl \quad \Gamma\vdash n{:}bool}{\Gamma\vdash (n\ ::){:}\text{f-type}} \qquad \dfrac{\Gamma\vdash f{:}\text{f-type} \quad \Gamma\vdash tl{:}bool\ list}{\Gamma\vdash f\ tl{:}bool\ list}}{\Gamma\vdash n{::}(f\ tl){:}bool\ list}}{\Gamma\vdash \text{if-exp}{:}bool\ list}$$

f-type:bool list → bool list
Γ = {n:bool, tl:bool list, f:bool list → bool list}

Figure 3. Hindley-Milner type inference

2 Compositional type inference

There have been many researches on helping programmers to locate and fix type errors. Among them, Chitil [1] proposed an interactive type debugger for Haskell in 2001. The main idea of his system is to employ compositional type inference (rather than Hindley-Milner type inference): the type of an expression is inferred solely from its subexpressions. Figure 2 shows the compositional inference tree for a part of the program in Figure 1.

In compositional type inference, the type of `f` is determined locally. For example, the type of `f` is at first an arbitrary type variable `c` in the tree. When it is applied to `tl`, it becomes `d` → `e`. When "`f tl`" is further placed in the wider context, the type of `f` is elaborated. If the programmer thinks that `f` should return a list of integers, he immediately sees that the if-expression (at the bottom of the tree) is wrong because the return type of `f` is (for the first time) `bool list` at this node.

This is in contrast to Hindley-Milner type inference where `f` has the same type throughout the program (Figure 3), making it harder to find why `f` has type `bool list` → `bool list`.

3 Algorithmic Debugging

To locate the source of an error, Chitil applies algorithmic debugging [2] on the type inference tree. Starting from the node of the type inference tree where a type error occurs, the debugger locates the source of an error by repeatedly asking the programmer if the typing judgement is correct according to his intention. For example, at the bottom of Figure 3, the programmer is asked if the if-expression has type `bool list` under the environment {`n:bool, tl:c,f:c→bool list`}. Because the programmer thinks that `f` should return `int list`, the answer is "no". After walking around the inference tree asking questions, the debugger locates the source of an error at the node whose premises are all correct, but the conclusion is not. In Figure 3, the error is at the if-expression because all of its premises are correct.

Chitil's system is attractive because the user is directed to the source of an error simply by answering questions. In this abstract, we report on our initial experience of using the type debugger (reimplemented for OCaml) in the introductory OCaml course in Ochanomizu University as well as the extension to the type debugger we have implemented.

4 User interface

Problems Although Chitil's framework is conceptually elegant, to use it in practice, we need to provide user interface. We also have to support not only the minimal language to show the principle of algorithmic debugging but also more practical subset of the language.

Solution We have implemented an Emacs interface to our type debugger, which highlights the part of the program the debugger is currently focusing on. The highlighted part changes as the debugger walks around the type inference tree. We have covered almost all the syntax required to program in the OCaml course (to program the shortest path problem for the Tokyo metro network), including type definition and exception handling.

5 Easier and reusable questions

Problems Chitil's original type debugger asks whether the whole typing is correct or not, as in:

```
if n then n :: (f tl) else (f tl) :
{n:bool, tl:c, f:c→bool list} ⊢ bool list
Are intended types an instance?
```

However, it is not easy for beginners to answer this question directly, because one has to determine if all the parts of the typing are correct. Furthermore, even if the answer was no, we do not obtain information on which part of the typing was wrong.

Solution We decompose each question into smaller pieces. We ask whether all the typings in the environment as well as the body of the typing are correct separately. Although the number of questions increases, it becomes much easier (especially for beginners) to answer. Furthermore, we can reuse the previous answers to automatically answer the same (or inferable) questions. We also provide an option to input the correct type instead of simply answering "no". The input type can be used to drastically reduce the number of questions.

6 Detailed error messages

Problems After the type debugger have located the source of a type error, the programmer must fix it. Because algorithmic debugging simply locates an expression in which a type error occurs, it is not immediately clear (especially for beginners) what was wrong. For example, in Figure 3, we are notified that the if expression is wrong, but to actually fix the problem, we have to find out that the predicate part was not a boolean.

Solution We have improved our type debugger to show detailed error messages. Since we have information on the intended types of variables through a series of decomposed questions, we can use it to infer precisely the source of errors. In the previous example, we can show that the predicate part of the if-expression was not type correct, because its type is not bool according to the programmer's intended type.

7 Experience and future work

Since most beginning students are not used to OCaml types, it is difficult for them to answer questions properly, even the decomposed questions described in Section 5. In this respect, the three extensions we have implemented were essential but not enough. We need more tailored questions and error messages. Still, some students find the debugger useful, after they are used to OCaml types.

According to the log of the debugger, more than two thirds of the errors were at function applications. They are further classified into simple type conflict between the types of functions and arguments, and the insufficient number of arguments. It turned out that our debugger can locate most of these type errors if the debugger is provided with the programmer's correct intended types (either from the programmer's response or from the purpose statements of functions which we utilize in our debugger). However, students sometimes do not understand their intention properly, and answer "yes" to all the questions!

One of the difficulties beginning students face is partial application. We plan to improve our type debugger to handle partial application specially, so that beginning students can easily follow what is going on. We also plan to interview students when they give wrong answers.

Acknowledgement. We would like to thank Olaf Chitil for explanation on his work and helpful comments.

References

[1] Chitil, O. "Compositional explanation of types and algorithmic debugging," *ICFP '01*, pp 193–204 (September 2001).

[2] Shapiro, E. Y. *Algorithmic Program Debugging*, Cambridge, MA: MIT Press (1983).