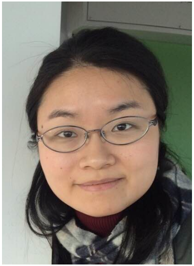
Principle and Practice of OCaml Type Debugger

Kenichi Asai



in collaboration with

← Kanae Tsushima and Yuki Ishii →

Ochanomizu University, Japan

September 18, 2016

Plan

What is an interactive type debugger?

Demo

1. Principle of an interactive type debugger in general.

OCaml type debugger, 1st version (2011)

2. How to scale it to a realistic language like OCaml.

OCaml type debugger, 2nd version (2013)

3. How to turn it into a practical tool.

OCaml type debugger, 3rd version (2014 -)

Future direction and summary

Demo (an exercise in introductory OCaml course)

Given a quadratic equation with integer coefficients $a(\neq 0)$, b, and c:

$$ax^2 + bx + c = 0$$

How many real solutions does the equation have?

The discriminant D is defined as:

$$D = b^2 - 4ac$$

Answer:

$$\mathsf{numRS}(a, b, c) = \begin{cases} 0 & (D < 0) \\ 1 & (D = 0) \\ 2 & (D > 0) \end{cases}$$

Demo (an exercise in introductory OCaml course)

numRS.ml

```
(* Purpose: given integer coefficients a, b, c,
   returns number of real solutions *)
(* numRS : int -> int -> int -> int *)
let numRS a b c =
  if discriminant a b c < 0 then "0"
  else if discriminant a b c = 0 then "1"
  else "2"

(* tests *)
let test1 = numRS 2 5 3 = 2
let test2 = numRS 5 4 3 = 0
let test3 = numRS 3 6 3 = 1
```

```
U:---  numRS.ml        Bot (15,27)     (Tuareg)
```

```
File "numRS.ml", line 15, characters 24-25:
Do you intend this expression to be of type: 'a -> 'a -> bool ?
(y/n/q) > y
File "numRS.ml", line 15, characters 12-23:
Do you intend this expression to be of type: string ?
(y/n/q) >
```

```
U:**-  *ocaml-toplevel*   Bot (16,10)     (Tuareg-Interactive:run
```

Interactive Type Debugger [Chitil2001]

- Interactive type debugger asks a series of questions to obtain programmer's intention.

In general, it is impossible to locate the source of a type error without knowing programmer's intention.

Type error consists of two conflicting types.

The source of the type error can be either of them (or somewhere in between).

- Using the answers, the type debugger navigates us through the source program.
- It identifies the source of a type error that is consistent with the programmer's intention.
- The final diagnosis changes, according to the answers.

Principle of Interactive Type Debugger

Algorithmic program debugging [Shapiro1983]:

- Compare the inferred type and programmer's intention.
- Detect their difference to locate the source of the type error.

Example: (fun x $\rightarrow$ x + x) "1"

The inferred type (in the OCaml compiler):

$$\dfrac{\dfrac{\dfrac{x : \text{int} \vdash x : \text{int} \quad x : \text{int} \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \quad x : \text{int} \vdash x : \text{int}}{x : \text{int} \vdash x + x : \text{int}}}{\vdash \text{fun } x \rightarrow x + x : \text{int} \rightarrow \text{int}} \quad \vdash \text{“1”} : \text{str}}{\vdash (\text{fun } x \rightarrow x + x)\ \text{“1”} : (\text{type error})}$$

cf. "str" stands for "string".

Scenario 1: the difference designates the source

The programmer thought “1” can be an integer.

The intention (in the programmer's mind):

$$\dfrac{\dfrac{\dfrac{x : \mathsf{int} \vdash x : \mathsf{int} \quad x : \mathsf{int} \vdash + : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \quad x : \mathsf{int} \vdash x : \mathsf{int}}{x : \mathsf{int} \vdash x + x : \mathsf{int}}}{\vdash \mathsf{fun}\ x \to x + x : \mathsf{int} \to \mathsf{int}} \qquad \vdash \text{“1”} : \color{red}{\mathsf{int}}}{\vdash (\mathsf{fun}\ x \to x + x)\ \text{“1”} : \mathsf{int}}$$

The inferred type (in the OCaml compiler):

$$\dfrac{\dfrac{\dfrac{x : \mathsf{int} \vdash x : \mathsf{int} \quad x : \mathsf{int} \vdash + : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \quad x : \mathsf{int} \vdash x : \mathsf{int}}{x : \mathsf{int} \vdash x + x : \mathsf{int}}}{\vdash \mathsf{fun}\ x \to x + x : \mathsf{int} \to \mathsf{int}} \qquad \vdash \text{“1”} : \color{red}{\mathsf{str}}}{\vdash (\mathsf{fun}\ x \to x + x)\ \text{“1”} : \color{blue}{(\text{type error})}}$$

Possible fix: $(\mathsf{fun}\ x \to x + x)\ \color{red}{1}$ $\leadsto 2$

Scenario 2: inferred derivation is not compositional

The programmer thought + is a string concatenation operator.

The intention (in the programmer's mind):

$$\dfrac{\dfrac{\dfrac{x : \mathrm{str} \vdash x : \mathrm{str} \quad x : \mathrm{str} \vdash + : \mathrm{str} \to \mathrm{str} \to \mathrm{str} \quad x : \mathrm{str} \vdash x : \mathrm{str}}{x : \mathrm{str} \vdash x + x : \mathrm{str}}}{\vdash \mathrm{fun}\ x \to x + x : \mathrm{str} \to \mathrm{str}} \quad \vdash \text{“1”} : \mathrm{str}}{\vdash (\mathrm{fun}\ x \to x + x)\ \text{“1”} : \mathrm{str}}$$

The inferred type (in the OCaml compiler):

$$\dfrac{\dfrac{\dfrac{x : \mathrm{int} \vdash x : \mathrm{int} \quad x : \mathrm{int} \vdash + : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \quad x : \mathrm{int} \vdash x : \mathrm{int}}{x : \mathrm{int} \vdash x + x : \mathrm{int}}}{\vdash \mathrm{fun}\ x \to x + x : \mathrm{int} \to \mathrm{int}} \quad \vdash \text{“1”} : \mathrm{str}}{\vdash (\mathrm{fun}\ x \to x + x)\ \text{“1”} : (\text{type error})}$$

The type of x becomes int, because x is an argument of +.

Scenario 2: use MGTT that is compositional

The programmer thought + is a string concatenation operator.

The intention (in the programmer's mind):

$$\dfrac{\dfrac{\dfrac{x : \mathsf{str} \vdash x : \mathsf{str} \quad x : \mathsf{str} \vdash + : \mathsf{str} \to \mathsf{str} \to \mathsf{str} \quad x : \mathsf{str} \vdash x : \mathsf{str}}{x : \mathsf{str} \vdash x + x : \mathsf{str}}}{\vdash \mathsf{fun}\ x \to x + x : \mathsf{str} \to \mathsf{str}} \qquad \vdash \text{“1”} : \mathsf{str}}{\vdash (\mathsf{fun}\ x \to x + x)\ \text{“1”} : \mathsf{str}}$$

The most general type tree (MGTT) (in the type debugger):

$$\dfrac{\dfrac{\dfrac{x : \alpha \vdash x : \alpha \quad x : \mathsf{int} \vdash + : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \quad x : \alpha \vdash x : \alpha}{x : \mathsf{int} \vdash x + x : \mathsf{int}}}{\vdash \mathsf{fun}\ x \to x + x : \mathsf{int} \to \mathsf{int}} \qquad \vdash \text{“1”} : \mathsf{str}}{\vdash (\mathsf{fun}\ x \to x + x)\ \text{“1”} : (\text{type error})}$$

Possible fix: $(\mathsf{fun}\ x \to x\ \hat{}\ x)\ \text{“1”} \rightsquigarrow \text{“11”}$

Principle of Interactive Type Debugger

Interactive type debugger:

- compares the inferred type and programmer's intention (obtained from the answers to questions),
- uses the most general type tree,
- detects the most specific difference, and
- reports it as the source of the type error.

OCaml Type Debugger, 1st version in 2011

- Direct implementation of the idea just shown.
- Written from scratch as an independent program.
- Construct the most general type tree.
- Tried in an introductory OCaml course in Ochanomizu Univ.

result:

- It could help students sometimes.
- But mostly, it was a failure.

reason: it does not scale.

- Limited support for the language constructs and types.
- Subtle deviation from the OCaml static semantics.

The principle of the interactive type debugger is simple and great.
However, the reality is different.

Making Principle to Scale

Although simple in principle, it is hard to construct the most general type tree precisely and for whole the OCaml language.

observations:

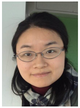
- The OCaml compiler performs type inference.
- The constructed type derivation tree in the OCaml compiler is not exactly what we want (i.e., the most general type tree).
- But they are *similar*!

The crucial question

Can we reuse the OCaml type inference for our purpose?

If we could (with small implementation efforts):

- We can support all the OCaml language constructs.
- No deviation from the OCaml static semantics.



Constructing MGTT using OCaml type inferencer

idea

Construct each node of the MGTT (most general type tree) one by one using OCaml type inferencer as a black box.

1. Decompose a given program into subprograms.
   - [(fun x → x + x) "1"] decomposes into [fun x → x + x] and ["1"].
2. Obtain their types using OCaml type inferencer.
   - [fun x → x + x] has type [int → int].
   - ["1"] has type [str].

result (so far):

$$\frac{\vdash [\text{fun x} \to \text{x} + \text{x}] : [\text{int} \to \text{int}] \qquad \vdash [\text{“1”}] : [\text{str}]}{\vdash [(\text{fun x} \to \text{x} + \text{x})\ \text{“1”}] : (\text{type error})}$$

Decomposition under Binder

1. Decompose a given program into subprograms.
   - [fun x → x + x] decomposes into fun x → [x + x].
2. Obtain their types using OCaml type inferencer.
   - fun x → [x + x] has type int → [int].

> To keep environment information, we represent a program by a focused expression (within [...]) with its context (outside [...]).

result (so far):

$$\dfrac{\dfrac{\vdash \text{fun x} \to [\text{x} + \text{x}] : \text{int} \to [\text{int}]}{\vdash [\text{fun x} \to \text{x} + \text{x}] : [\text{int} \to \text{int}]} \quad \vdash [\text{“1”}] : [\text{str}]}{\vdash [(\text{fun x} \to \text{x} + \text{x})\ \text{“1”}] : (\text{type error})}$$

Decomposition with Context

1. Decompose a given program into subprograms.
   - fun x → [x + x] decomposes into fun x → [x], fun x → [+], and fun x → [x].
2. Obtain their types using OCaml type inferencer.
   - fun x → [x] has type $\alpha \to [\alpha]$.
   - fun x → [+] has type $\alpha \to [\text{int} \to \text{int} \to \text{int}]$.
   - fun x → [x] has type $\alpha \to [\alpha]$.

> We keep the context, and decompose the focused expression.

resulting MGTT (omitting fun x → [x] at the top right):

$$\dfrac{\dfrac{\dfrac{\vdash \text{fun x} \to [\text{x}] : \alpha \to [\alpha] \qquad \vdash \text{fun x} \to [+] : \alpha \to [\text{int} \to \text{int} \to \text{int}]}{\vdash \text{fun x} \to [\text{x} + \text{x}] : \text{int} \to [\text{int}]}}{\vdash [\text{fun x} \to \text{x} + \text{x}] : [\text{int} \to \text{int}]} \qquad \vdash [\text{“1”}] : [\text{str}]}{\vdash [(\text{fun x} \to \text{x} + \text{x})\ \text{“1”}] : (\text{type error})}$$

OCaml Type Debugger, 2nd version in (2012-)2013

- Implemented using the OCaml type inferencer (ver 3.12.1).
- Supports most of the constructs, incl. exceptions, modules, ...
- Faithful to the OCaml static semantics.
- Preliminary version in 2012, fully functional in 2013.

result:

- First usable (reliable) version in practice.
- Reveals a lot of issues. It was not a practical tool yet.

main issues:

- The wording of questions (very) important.
  - Bad "Is this expression of type int?"
  - Good "Do you intend this expression to be of type int?"
- More detailed error explanation required.

Type Error Explanation

The type debugger explained so far designates one expression as the source of a type error. But it does not say why.

Suppose the source of the type error is located at an if expression.

if 0 then 1 else 2

Bad Something is wrong in this expression.

Good The predicate part '0' has type int, but it must be bool.

if true then 1 else "2"

Good The then part '1' has type int and else part "2" has type string, but they must be equal.

Type Error Explanation

For each identified expression, we want to present the violated type constraint.

- The algorithmic program debugging works for any languages.
- To show violated type constraints, OCaml-specific handling is required.

We logged all the interaction students had with the type debugger. For all the interaction taken in 2012, we manually

- classified them according to the kind of errors,
- analyzed them if type debugger worked fine, and
- provided better error messages. E.g.,
  - Which branch of match expression was wrong.
  - Which argument of function application was wrong.

OCaml Type Debugger, 3rd version in 2014 -

- Implemented in 2014 with OCaml ver 4.01.0
  and in 2015 with OCaml ver 4.02.1.
- Easy to port; we don't have to re-implement type inference.
- Stable enough; in our class, it is launched automatically whenever type error occurs.

result:

- Works well!
  - "It helped me a lot finding how my intention was not reflected in the program."
  - "I could find bugs by myself using the type debugger."
  - "I was naturally led to think about types."
- It is hard to answer properly at the beginning, but they learn.
- Naturally, new issues arise.

Future Direction and Summary

- How to handle syntax errors.
- Reduce the number of questions.
- Utilization of type information found in comments.
- Enhance user interface.
- Automatic log analysis.
- Testing framework of type debugger (and interactive programs).

OCaml type debugger:

`http://pllab.is.ocha.ac.jp/~asai/TypeDebugger/`

Principle applied to practice with a simple idea pushed through