A Weighted Type Error Slicer

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

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

Extended abstract submitted to IFL2013

Abstract. This extended abstract presents how to make weighted type error slices. In standard type error slices, each part of a type error slice looks to be at the same level as far as the source of the type error goes. However, when a type error slice is large, to search the source of the type error from it is hard work. To this end, we extend type error slices with the *weights* that mean the likelihood of each expression being the source of a type error. We use majority vote of subexpressions and obtain which subexpression looks to be the source of the type error. Using weights we can narrow the area of a program to search the source of the type error and reduce the burden of programmers for type debugging.

1 Overview

When a compiler returns a type error message, programmers often have to search the source of the type error by hand. To do that, type error slices are useful for programmers, because type error slices narrow the area for type debugging. A type error slice includes all parts that relate to one type conflict. Therefore we can surely find the source of a type error in a type error slice. To explain what is a type error slice, let us consider the following ill-typed program written in OCaml:

```
let f n lst = List.map (fun x -> x ^ n) lst in
 (f 2.0 [3.0; 4.0])
```

In this program, the types of `2.0` and `^` conflict. Because this program contains parts that require these two conflicting types to be the same type, type error occurs. In this program, we passed `2.0` (of type `float`) to the function `f` as the first argument. The type of the first argument of `f` is forced to be `string` through application of `^` (in `x ^ n`). This flow of unifying types contributes the type conflict between `2.0` and `^`. If two conflicting types and the flow of unifying the two types appear in the program, surely type error occurs. The following program represents a type error slice of the previous example:

```
let f n ... = ... (fun ... -> ... ^ n) ... in
(f 2.0 ...)
```

The parts abstracted by "`...`" do not contribute to this type conflict. Since this slice includes all the parts related to the type conflict, programmers can locate the source of the type error by debugging only this type error slice. As described above, type error slices can reduce the burden on programmers. However, if the original program is huge, its slice could be large.

To overcome this problem, we want to know which part is likely the source of the type error. When we look at a type error slice of an ill-typed program, we might conclude that each part has an equal chance of being the source of the type error. For example, in the previous slice, `2.0` and `^` look to be at the same level as far as the source of the type error goes. This conclusion, however, is not necessarily true as implied by the following observation. In the previous example, we can consider another slice:

```
let f ... lst = List.map (fun x -> x ^ ...) lst in
(f ... [3.0; 4.0])
```

The point here is that this slice includes "`^`" but not "`2.0`." To see this clearly, let us consider the following slice:

```
let f ... ... = ... (fun ... -> ... ^ ...) ... in
(f ... ...)
```

This slice is the intersection of the previous two slices. Because this slice is well-typed, it is not a type error slice. Although this slice may not include the source of the type error, it does include suspicious parts of the source. In this example `^` is suspicious to be the source of the type error, because it is included in both the slices. This observation about the intersection of several type error slices suggests that each part of a type error slice has a different chance of being the source of the type error.

2 Our approach

The intersections of type error slices are very intuitive and produce good results. However, to obtain such intersections, we have to obtain the type error slices first. The computational complexity needed to obtain a type error slice of an ill-typed program is $O(n^2)$, where n is the size of the program. Furthermore, to obtain all slices, we have to repeat this calculation $n!$ times. Thus, the computational complexity needed to obtain an intersection for a large program would be large. We thus need a way to reduce this cost and in the following, we introduce an approach to obtain the likelihood of each expression being the source of a type error.

2.1 Brief overview

Let us consider the program `[1;2;true]`. Because two elements of this list are numbers and one element is a boolean, we expect the minority `true` is wrong.

This is the key point of our approach. The problem is how we can obtain such information. The main ideas that we will exploit are abstraction of programs and majority vote.

First, we abstract one part of the program and infer its type. The result of doing this for the above example is shown in the table below.

abstracted program	well-typed?
`[1; 2; ..]`	○
`[1; ..; true]`	×
`[..; 2; true]`	×

If an abstracted program is ill-typed, its subprograms may contribute to the type error. For example, because `[1; ..; true]` is ill-typed, its subprograms `1` and `true` contribute to the type error. Therefore, we count the number of contributions of each subprogram. The following table is the result of doing so.

abstracted program	contributions
`1`	1
`2`	1
`true`	2

This table shows us the likelihood of each expression being the source of a type error. From this, we know that `true` has a higher probability of being the source of the type error than `1` or `2` has. This result is what we anticipated.

We apply this idea to simply typed lambda-calculus extended with let polymorphism. To obtain better weights, we extend this idea to use the context that does not contain type error slices.

2.2 The point and contribution

Our approach has two main points. One is that we use the compiler's type inferencer to construct a type error slicer. Many type error slicers [1] use a tailor-made type unification. Although it has a certain flexibility, its results may not correspond to those of the compiler's type inferencer, and it has low scalability. In contrast, by using the compiler's inferencer we can make a type error slicer that has maintainability and high scalability. This contribution is the same with Schilling's approach [2].

The other point is that we can obtain *weighted* type error slices. The weights are the likelihoods of the expressions being the source of a type error. They can help programmers to reduce their burden during debugging.

References

1. Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP'03)*, Volume 50 Issue 1-3 (2004).
2. Schilling, T. "Constraint Free Type Error Slicing," *Proceedings of the 12th international conference on Trends in Functional Programming (TFP'11)*, pp. 1–16 (2012).