Finding Minimum Type Error Sources

Zvonimir Pavlinovic, Tim King, and Thomas Wies
New York University
{pavlinovic,taking,wies}@cs.nyu.edu

Abstract: Automatic type inference is a popular feature of functional programming languages. If a program cannot be typed, the compiler reports the first location with a type conflict as the source of the error. However, this is often not the true error source. We present a general algorithm that helps to produce more meaningful type error reports. The algorithm finds all minimum error sources, where the definition of minimum is given by a compiler-specific ranking criterion (e.g. minimizing the program changes needed to fix the error). The approach works by reducing the search for minimum error sources to an optimization problem that we formulate in terms of weighted maximum satisfiability modulo theories (MaxSMT). This formulation allows us to build on SMT solvers to support rich type systems and, at the same time, abstract from the concrete ranking criterion. We have implemented an instance of our algorithm targeted at OCaml programs. Our evaluation shows that our approach significantly improves upon the quality of type error reports produced by state of the art compilers. This work appeared at OOPSLA 2014 where it won a Best Paper Award.

1 Problem

In functional programming languages such as OCaml and Haskell, programmers are not obliged to provide type annotations. Nevertheless, these languages guarantee strong static typing by automatically inferring types based on how expressions are used in the program. Unfortunately, the convenience of type inference comes at a cost: if the program cannot be typed, the compiler-generated error message often does not help to fix the error. Confusing error messages increase the debugging time and make it more difficult for novice programmers to learn the language. In this paper, we present a general framework for producing more meaningful type error messages.

Typical type inference algorithms report type errors on the fly. If the inferred type of the current program expression conflicts the inferred type of its context, the inference algorithm immediately stops and reports an error at the current program location. Although fast in practice, this approach also produces poor error diagnostics. In particular, it might be the case that the programmer made a mistake with the previous usages of the offending expression, or with some other related expressions. For example, consider the following simple OCaml program:

```ocaml
1 type 'a lst = Null | Cons of 'a * 'a lst
2 let x = Cons(3, Null)
3 let _ = print_string x
```
The standard OCaml compiler reports a type mismatch error for expression $x$ on line 3, as the code before that expression is well typed. However, perhaps the programmer defined $x$ incorrectly on line 2 or misused the `print_string` function. As $x$ is defined just before the error line, it seems more likely that the error is caused by a misuse of `print_string`. In fact, the student author of this code confirmed that this is the real source of the error. This simple example suggests that in order to generate useful error reports compilers can consider several possible error causes and rank them by their relevance. Hence, there is a need for an infrastructure that can supply compilers with error sources that best match their relevance criteria. In this work, we propose a general algorithm based on constraint solving that provides this functionality.

2 Approach

Unlike typical type inference algorithms, we do not simply report the location of the first observed type inconsistency. Instead, we compute all minimum sets of expressions each of which, once corrected, yields a type correct program. Compilers can then use these computed sets for generating more meaningful error reports or even for providing automatic error correction. The considered notion of minimum is controlled by the compiler. For example, the compiler may only be interested in those error causes that require the fewest changes to fix the program.

The crux of our approach is to reduce type error localization to the weighted maximum satisfiability modulo theory (MaxSMT) problem. Specifically, our algorithm builds on existing work that rephrases type inference in terms of constraint solving. Each program expression is assigned a type variable and typing information is captured in terms of constraints over those variables. If an input program has a type error, then the corresponding set of typing constraints is unsatisfiable. We encode the compiler-specific ranking criterion by assigning weights to the generated typing constraints. A weighted MaxSMT solver then computes the satisfiable subsets of the constraints that have maximum cumulative weight. As constraints directly map to program expressions, the complements of these maximum sets represent minimum sets of program expressions that may have caused the type error.

The use of SMT solvers has several additional advantages. First, it allows support for a variety of type systems by instantiating the MaxSMT solver with an adequate reasoning theory. Typing constraints for Hindley-Milner type systems can be encoded in the theory of inductive data types. More complex type systems such as liquid types may involve additional reasoning theories (e.g., arithmetic). Second, the framework does not introduce a substantial implementation overhead since the SMT solver can be used as a black box.

We have implemented an instance of our framework targeted at Hindley-Milner type systems and evaluated it on existing OCaml benchmarks for type error localization. Our experiments suggest that our approach can produce minimum sources of type errors subject to useful ranking criteria. Already with the relatively simple ranking criterion that we used in our experiments, we observed that our algorithm yields a better detection rate than OCaml’s type checker.