

Finding Minimum Type Error Sources

Zvonimir Pavlinovic, Tim King, and Thomas Wies

New York University

Abstract. Automatic type inference is a popular feature of functional programming languages. If a program cannot be typed, the compiler typically reports a single program location in its error message. This location is the point where the type inference failed, but not necessarily the actual source of the error. Other potential error sources are not even considered. Hence, the compiler often misses the true error source, which increases debugging time for the programmer. In this paper, we present a general framework for automatic localization of type errors. Our algorithm finds all minimum error sources, where the exact definition of minimum is given in terms of a compiler-specific ranking criterion. Compilers can use minimum error sources to produce more meaningful error reports, and for automatic error correction. Our approach works by reducing type inference to constraint satisfaction. We then formulate the problem of computing minimum error sources in terms of weighted maximum satisfiability modulo theories (MaxSMT). Ranking criteria are incorporated by assigning weights to typing constraints. The reduction to MaxSMT allows us to build on decision procedures to support rich type systems.

1 Introduction

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. Consequently, confusing error messages increase the debugging time. In this paper, we present a general framework for producing more meaningful type error messages.

A typical type inference algorithm immediately stops and reports an error at the current program location if the inferred type of the current program expression conflicts the inferred type of its context. Although fast in practice, this approach also produces poor error diagnostics. For example, consider the following simple OCaml program taken from the student benchmarks in [4]:

```
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. The student author of this code confirmed that the latter 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. In this work, we propose a general algorithm based on constraint solving that supplies compilers with error sources best matching their relevance criteria.

2 Overview of the 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. The considered notion of minimality 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 maximum satisfiability modulo theory (MaxSMT) problem. 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.

We explain our reduction using the following OCaml program as an example:

```
let x = "hi" in not x
```

Clearly, the program is not well typed as the operation `not` on Booleans is applied to a variable `x` of type `string`. Our constraint generation procedure takes the program and generates a set of typing constraints using the OCaml type inference rules. For our example program, the constraint generation produces the following set of assertions:

$$\alpha_{not} = \text{fun}(\text{bool}, \text{bool}) \quad [\text{Def. of not}] \quad (1)$$

$$\alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \quad \text{not } x \quad (2)$$

$$\alpha_{app} = \alpha_{not} \quad \text{not} \quad (3)$$

$$\alpha_i = \alpha_x \quad x \quad (4)$$

$$\alpha_x = \text{string} \quad x = \text{"hi"} \quad (5)$$

Each assertion comes from a particular program expression shown to the right of the assertion. For instance, the assertion (1) is generated from the definition of the function `not` in OCaml's standard library. It specifies the type α_{not} of `not` as a function type from `bool` to `bool`. The generated type constraint is interpreted in the theory of inductive data types, where type variables stand for variables and other expressions, like

`fun` and `bool`, are injective constructors. The generated type constraint is unsatisfiable, confirming that there is a type error.

It is easy to see that removing one assertion from the generated typing constraints makes the remaining set of assertions satisfiable. The expression corresponding to a removed constraint is regarded as an error source, i.e., correcting that expression makes the whole program well typed. In general, an *error source* is a set of program expressions that, once corrected, yield a well typed program. A *minimal error source* is an error source such that none of its proper subsets is also an error source. In our running example, each program expression is a minimal error source.

Compilers incorporate ranking criteria by assigning weights to program expressions. Smaller weights indicate that the corresponding expression is more likely contributing to the type error. Given a ranking criterion, a *minimum error source* is an error source with a minimum cumulative weight (i.e., it is also minimal). For example, consider a ranking criterion that assigns to each program expression the weight equal to the expression size in its abstract syntax tree form. Then, the expression corresponding to the assertion (2) is not a minimum error source as it has weight 2, while other minimal error sources have weight 1.

To find a minimum error source subject to a given a ranking criterion, our constraint generation procedure propagates weights from expressions to associated assertions. Then, we use a weighted MaxSMT procedure to compute a maximum satisfiable subset of these assertions. The program expressions that correspond to the complement of these assertions constitute the minimum error source. We have implemented this algorithm and applied it to the OCaml benchmarks from [4]. Our experiments showed that our approach can find minimum error sources subject to useful ranking criteria. For a detailed discussion of the algorithm and implementation we refer the reader to the full paper, which is available at http://cs.nyu.edu/wies/publ/finding_minimum_type_error_sources.pdf.

Complexity and Tractability. The decision problem that asks whether a given program is well-typed with respect to the Hindley-Milner type system is EXPTIME-complete [5,3]. Nevertheless, actual implementations of type checkers for OCaml and other languages that are based on this type system achieve good performance in practice. This is possible because type checking can be done compositionally by computing principle types using most general unifiers [6]. As explained above, we reduce the complement of the type checking problem for Hindley-Milner to satisfiability modulo the theory of inductive data types. This reduction preserves the complexity of the problem. However, a naive handling of polymorphism in the constraint generation results in an exponential explosion in the size of the generated constraints. This explosion quickly leads to intractable performance, even for moderately sized programs.

The question is then how this exponential explosion can be deferred to take advantage of the heuristics in the SMT solver that prune the search space and achieve good performance in practice. In the extended version of this abstract, we discuss a possible answer to this question. Our approach combines two ideas. The first idea is to use an encoding of type constraints for polymorphic functions that is based on stratified quantified constraints. The resulting constraints are linear in the size of the input program and remain decidable. The second idea is to make the optimistic assumption that

most let-bound variables do not contribute to minimum type error sources. This idea leads to an iterative algorithm in which the types of let-bound variables are summarized by their principle types, i.e., the most general solutions of the associated typing constraints. Only if such a most general type occurs in a minimum type error source do we expand its associated constraints and reiterate. While we have not yet implemented this improved algorithm, we are confident that it will achieve considerable performance improvements in practice.

3 Related Work and Conclusions

Closely related to our approach is the *Seminal* [4] tool, which computes several possible error sources by repeated calls to the type checker. However, the search for error causes is based on heuristics and provides no formal guarantees that all error sources are found, respectively, that they are ranked according to some criterion. Zhang and Myers [8] encode typing information for Hindley-Milner type systems in terms of constraint graphs. The generated graphs are then analyzed to find most likely error sources by using Bayesian inference. It is unclear how this approach would support more expressive type systems. Previous approaches based on constraint solving [1,7] produce minimal but not minimum error sources and consider specific ranking criteria for specific type systems. Our approach is in part inspired by the *Bug-Assist* tool [2], which uses a MaxSAT procedure for fault localization in imperative programs. However, the problem we are solving is quite different.

In summary, we propose a novel framework for type error localization based on constraint solving. Our framework enables compilers to search for error sources of particular interest and supports rich type systems by relying on SMT solvers.

References

1. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, pages 189–224, 2004.
2. M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV*, pages 504–509. Springer-Verlag, 2011.
3. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML Typability is DEXTIME-Complete. In *CAAP*, pages 206–220, 1990.
4. B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI*. ACM Press, 2007.
5. H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *POPL*, pages 382–401, 1990.
6. J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.
7. P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91. ACM, 2004.
8. D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *POPL*, pages 569–581. ACM, 2014.