Finding Minimum Type Error Sources

Zvonimir Pavlinovic
New York University
zvonimir@cs.nyu.edu

Tim King
New York University
taking@cs.nyu.edu

Thomas Wies
New York University
wies@cs.nyu.edu

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 the search for minimum error sources to an optimization problem that we formulate in term of weighted maximum satisfiability modulo theories (MaxSMT). The reduction to weighted MaxSMT allows us to build on SMT solvers to support rich type systems and at the same time abstract from the concrete criterion that is used for ranking the error sources. 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 evaluation shows that our approach has the potential to significantly improve the quality of type error reports produced by state-of-the-art compilers.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Diagnostics; F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords Type Errors, Diagnostics, Satisfiability Modulo Theories

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10...$15.00.
http://dx.doi.org/10.1145/2660193.2660230

1. Introduction

In functional programming languages such as OCaml [31] and Haskell [18], 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 [24]. 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 taken from the student benchmarks in [24]:

```
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 [31] 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.

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 [1, 32, 38]. 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 [19, 29] can be encoded in the theory of inductive data types [4]. More complex type systems such as liquid types [34] 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.

Implementation. We have implemented an instance of our framework targeted at Hindley-Milner type systems. Our implementation builds on top of the EasyOCaml [14] system, the SMT solver CVC4 [2], and the SAT solver Sat4j [23], which also supports weighted partial MaxSAT. We have evaluated our implementation on existing OCaml benchmarks [24] for type error localization. Our experiments suggest that our approach can produce minimum sources of type errors subject to useful ranking criteria. Also, we evaluated the effectiveness of our algorithm in identifying the true error source by comparing it against the error diagnostics of the OCaml type checker. 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.

Related Work. Previous work on localization of type errors has mainly focused on designing concrete systems for generating quality type error messages. Existing approaches range from showing a relevant portion of a failed type inference trace [12, 41], a program slice involved in the error [15, 39], to specially crafted type systems [7, 8, 30]. More closely related to our approach is the `Seminal` [24] 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 [42] 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. On the other hand, the proposed technique is designed for general diagnosis of static errors, whereas we focus specifically on diagnosis of type errors. Previous approaches based on constraint solving [17, 37] 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 [20], which uses a MaxSAT procedure for fault localization in imperative programs. However, the problem we are solving in this paper is quite different.

Contributions. Our contributions can be summarized as follows:

- We present a novel framework for producing quality type error diagnostics. Given a supplied ranking criterion, the framework generates minimum type error sources.
- By reducing type error localization to SMT constraint solving, our framework supports a variety of type systems without introducing substantial complexity to existing language implementations.
- We have implemented our algorithm for Hindley-Milner type systems and applied it to the OCaml benchmarks used in [24].

2. Overview

In this section, we provide an overview of our approach and explain it through several illustrative examples.

The high-level execution flow of our constraint-based type error localization is shown in Figure 1. The framework can be viewed as a compiler plug-in. When type inference takes place, our algorithm starts by generating typing assertions for the given input program. The constraint generation incorporates a compiler-specific ranking criterion by appropriately assigning weights to the assertions. After the constraint generation finishes, the produced annotated assertions, constituting a typing constraint, are passed to a weighted MaxSMT solver. If the input program has a type error, the generated constraint is unsatisfiable. In this case,

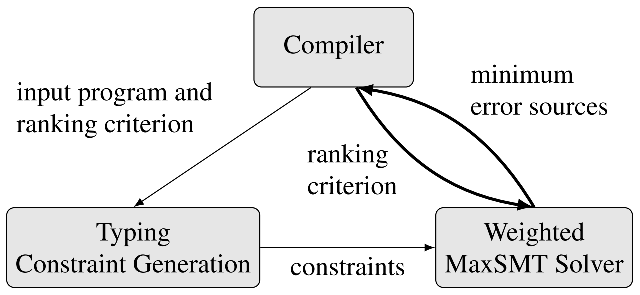

Figure 1. High-level overview of constraint-based type error localization. Thick arrows represent a looping interaction between a compiler and the SMT solver.

the MaxSMT solver finds all error sources that are minimum subject to the specified ranking criterion. The compiler can then iteratively interact with the solver to further rank and filter the error sources to generate an appropriate error message. This way, our framework can support interaction with the programmer. In particular, the compiler can take feedback from the programmer and pass it to the solver to guide the search for the error source the programmer needs. In the following, we describe the actual algorithm in more detail and highlight the main features of our approach.

2.1 Minimum Error Sources

First, let us make the notion of minimum error source more precise. An error source is a set of program locations that need to be fixed so that the erroneous program becomes well typed. Usually, not all error sources are equally likely to cause the error, and the compiler might prefer some error sources over others. In our framework, the compiler provides a criterion for ranking error sources by assigning weights to program locations. Our algorithm then finds the error sources with minimum cumulative weight. As an example, consider the following OCaml program:

```
let f x = print_int x in
let g x = x + 1 in
let x = "hi" in
  f x;
  g x
```

Note that the program cannot be typed since the functions f and g expect an argument of type int, but both are applied to x whose type is string. The program has several possible error sources. One error source is the declaration of x. Namely, changing the declaration of x to, say, an integer constant would make the program well typed. Another error source is given by the two function applications of f and g. For example, replacing both f and g by print_string would yield another well-typed program.

Now consider a ranking criterion that assigns each program location equal weight 1. Then the cumulative weight of the first error source is 1, while the weight of the second error source is 2. Our algorithm would therefore report the first error source and discard the second one. This simple ranking criterion thus minimizes the number of program locations that need to be modified so that the program becomes well typed. Later in this section, we discuss more elaborate ranking criteria that are useful for error localization.

2.2 Reduction to MaxSMT

Next, we show how the reduction to weighted MaxSMT works through an example. Consider the following OCaml program:

```
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 assertions using the OCaml type inference rules. This set of assertions constitutes a typing constraint. A formal description of the constraint generation can be found in Section 4.1. For our example program, the constraint generation produces the following set of assertions:

$$\begin{aligned} \alpha_{not} &= \mathsf{fun}(\mathsf{bool}, \mathsf{bool}) && \text{[Def. of \texttt{not}]} && (1)\\ \alpha_{app} &= \mathsf{fun}(\alpha_i, \alpha_o) && \texttt{not x} && (2)\\ \alpha_{app} &= \alpha_{not} && \texttt{not} && (3)\\ \alpha_i &= \alpha_x && \texttt{x} && (4)\\ \alpha_x &= \mathsf{string} && \texttt{x = "hi"} && (5) \end{aligned}$$

The constraint encoding preserves the mapping between assertions and associated program locations. That is, each assertion comes from a particular program expression shown to the right of the assertion. The assertion (1) is generated from the definition of the function not in OCaml's standard library [25]. It specifies the type α_{not} of not as a function type from bool to bool. The assertions (2) to (4) specify the types of all subexpressions of the expression not x in the program. Finally, the assertion (5) specifies the type of x, according to its declaration.

The generated typing constraint is interpreted in the theory of inductive data types, which is supported by many SMT solvers [2, 11, 13]. In this theory, the terms α_{app}, α_i, α_o, α_{not}, and α_x stand for type variables, while the terms bool and string are type constants. The function fun is a type constructor that maps an argument type α_i and a result type α_o to a function type from α_i to α_o. The theory interprets fun as an injective constructor. Hence, the assertions (1) to (3) together imply $\alpha_i = \mathsf{bool}$. Type constants such as bool and string are interpreted as pairwise distinct values. The assertions (4) and (5) imply $\alpha_i = \mathsf{string}$, we conclude $\mathsf{bool} = \mathsf{string}$ together with the assertions (1) to (3) – a contradiction. Consequently, the generated typing constraint is unsatisfiable, confirming there is a type error.

As in the previous example, we first assume a ranking criterion that assigns each program location equal weight 1.

The problem of finding the minimum error sources in the program is then encoded into a weighted MaxSMT problem by assigning weight 1 to each assertion in the generated constraint. The weighted MaxSMT solver computes all subsets of assertions in the typing constraint that are satisfiable and have maximum cumulative weight. Thus, the complements of those sets encode the minimum error sources. In Section 4.2, we discuss the weighted MaxSMT procedure that we use in our implementation in detail.

For the running example, there are altogether five complement sets of maximum satisfiable subsets, each consisting of a single assertion (1)-(5). Removing any single assertion leaves the remaining assertions satisfiable. In other words, by fixing the program expression associated with the removed assertion, the input program becomes well typed. For example, consider the assertion (3). Removal of that assertion can be interpreted as using a different function instead of `not` in the application `not x`. Removal of the assertion (1) can be seen as changing the internals of `not` so that it applies to string values. Hence, the meaning associated with removing a certain assertion can be utilized for suggesting possible error fixes.

2.3 Ranking Criteria

Not all error sources are equally useful for fixing the type error. In the previous example, it is unlikely that the true error source is located in the implementation of the function `not`, since it is part of the standard library. Compilers might want to eliminate such error sources from consideration. Similarly, the error source corresponding to the removal of assertion (2) implies that the programmer should use some other expression than `not x` in the program. This error source seems less specific than the remaining error sources, so compilers might want to rank it as less likely. In general, compilers might want to rank error sources by some definition of usefulness. In our framework, such rankings are encoded by assigning appropriate weights to program locations, and hence assertions in the generated typing constraint.

Hard Assertions. One way of incorporating ranking criteria is to specify that certain assertions must hold, no matter what. Such assertions are commonly referred to as *hard assertions*. Assertions that the MaxSMT solver is allowed to remove from the assertion set are called *soft*. The compiler may, for instance, annotate all assertions as hard that come from the definitions of functions provided in the standard library. This would encode that all type error sources must be located in user-defined functions. If we apply this criterion to our previous example, we annotate the assertion (1) as hard. This eliminates the error source implying that the implementation of `not` must be modified. Other assertions that might be considered hard are assertions that come from user-provided type annotations in the program.

Weighted Clauses. Our approach supports more sophisticated ranking criteria than distinguishing only between hard and soft assertions. Such criteria are encoded by assigning weights to assertions. Going back to our running example, compilers might favor error sources consisting of smaller program expressions. For instance, each assertion can be assigned a weight equal to the size of the corresponding expression in the abstract syntax tree. This way, `not x` in our previous example is not reported, as desired.

To demonstrate the power of weighted assertions, we consider another example from the student benchmarks in [24], which served as the motivating example for the type error localization technique presented in [42]:

```
1  let f(lst:move list):
2      (float*float) list = ...
3    let rec loop lst x y dir acc =
4      if lst = [] then
5        acc
6      else
7        print_string "foo"
8    in
9    List.rev
10     (loop lst 0.0 0.0 0.0 [(0.0,0.0)] )
```

The standard OCaml compiler reports a type error in the shaded expression on line 10. However, the actual error cause lies in the misuse of the `print_string` function on line 7, as reported by the authors of [42]. The technique of Zhang and Myers [42] correctly reports the expression on line 7 as the most likely error cause. With the ranking criteria that we introduced so far, our algorithm generates two error sources that both have minimum weight. These error sources can be interpreted as follows:

1. Replace the function `print_string` on line 7.
2. Replace the function `loop` on line 10.

However, the second error source is not likely the actual cause of the error: using some other function than `loop` on line 10 would mean that the `loop` function is not used at all in the program. In fact, the OCaml compiler would generate a warning for the fixed program.

The compiler can thus analyze the produced error sources and find those sources that correspond to the removal of entire functions from the program. Assertions associated with such error sources can then be set as hard. In the running example, the solver will then produce just a single error source, indicating the actual error cause.

This additional ranking criterion can also be encoded directly in the typing constraint without a second round of interaction with the solver. Suppose the program contains a user-defined function `f` whose type is described by a type variable α_f. Suppose further that `f` is used n times in the program and for i, $1 \leq i \leq n$, let the type variable α_i indicate the type imposed on `f` by the context of the i-th

```
 1 class p x_init =
 2  object
 3    val mutable x = x_init
 4    method get = x
 5    method move y = x <- x +. y
 6  end
 7
 8 class ap x_init =
 9  let origin = x_init / 10 * 10 in
10  object
11    val mutable x = x_init
12    method get = float_of_int x
13    method offset = x - origin
14    method move y = x <- x + (y mod 10)
15  end
16
17 let coerce x = (x: ap :> p)
18
19 let x = new ap 10.0
20 let _ = (coerce x)#move 4.5
```

Figure 2. OCaml program with invalid subtype coercion

usage of `f`. Then we can add the following additional hard assertion to the typing constraint: $\alpha_1 = \alpha_f \vee \cdots \vee \alpha_n = \alpha_f$. This assertion expresses that `f` needs to be used at least once, effectively encoding the additional criterion.

2.4 Supporting Expressive Type Systems

One of the advantages of our algorithm compared to existing type localization techniques is that it generalizes to more expressive type systems. All we need to do is augment the MaxSMT solver with the appropriate reasoning theories to support richer typing constraints. We discuss one such extension in more detail.

Subtyping. All of the examples that we have considered so far belong to the Caml subset of the OCaml language. This subset supports polymorphic higher-order functions, tagged unions, and records, but no classes and objects. The theory of inductive data types is sufficient for expressing the typing constraints for Caml. To support the object-oriented features of OCaml, we can incorporate additional reasoning theories.

As a motivating example, consider the OCaml program shown in Figure 2. The program declares two classes: a class `p` representing a point on a continuous line, and a class `ap` that represents a point on a line adjusted to the closest multiple of 10. Each class implicitly defines an *object type* of the same name as the class. An object type associates each method of an object with the type of that method. The object type `p` is

```
<get:float, move:float->unit>
```

and the object type `ap` is

```
<get:float, offset:int, move:int->unit>
```

OCaml uses structural subtyping to relate object types. That is, an object type s is a subtype of an object type t iff (1) s has all the methods of t and (2) for each method m of t, the type of m in s is a subtype of the type of m in t. Method subtyping is defined as expected. In particular, for `a -> b` to be a subtype of `c -> d`, the type `c` must be a subtype of `a` and the type `b` must be a subtype of `d`. For non-object types, the subtype relation reduces to type identity. For example, in the previous program, the object type `ap` is not a subtype of `p` because the argument type of `ap`'s `move` method expects an `int` instead of a `float`.

In OCaml, subtyping cannot be used implicitly. Instead it must be enforced explicitly via subtype coercion. In our example program, such a coercion can be seen on line 17. This coercion is invalid since `ap` is not a subtype of `p`. As we show below, our algorithm can report several locations as possible error sources.

Subtyping Assertions. For brevity, we focus on the assertions that are generated from the coercion constraint on line 17 as well as the relevant assertions that come from the two class declarations:

$$\{\mathtt{get}, \mathtt{move}\} \subseteq \{\mathtt{get}, \mathtt{offset}, \mathtt{move}\} \quad \text{[line 17]} \quad (6)$$

$$\beta_{\mathtt{ap\#get}} :> \beta_{\mathtt{p\#get}} \quad \text{[line 17]} \quad (7)$$

$$\alpha_{\mathtt{p\#move}} :> \alpha_{\mathtt{ap\#move}} \quad \text{[line 17]} \quad (8)$$

$$\rho_{\mathtt{ap\#move}} :> \rho_{\mathtt{p\#move}} \quad \text{[line 17]} \quad (9)$$

$$\beta_{\mathtt{p\#get}} = \mathsf{float} \quad \text{[line 4]} \quad (10)$$

$$\beta_{\mathtt{ap\#get}} = \mathsf{float} \quad \text{[line 12]} \quad (11)$$

$$\alpha_{\mathtt{p\#move}} = \mathsf{float} \quad \text{[line 5]} \quad (12)$$

$$\rho_{\mathtt{p\#move}} = \mathsf{unit} \quad \text{[line 5]} \quad (13)$$

$$\alpha_{\mathtt{ap\#move}} = \mathsf{int} \quad \text{[line 14]} \quad (14)$$

$$\rho_{\mathtt{ap\#move}} = \mathsf{unit} \quad \text{[line 14]} \quad (15)$$

The encoding uses pairwise distinct constants `get`, `offset`, and `move` to represent the method identifiers. The assertions (6)-(9) encode the coercion constraint on line 17. The remaining assertions come from the indicated method declarations. Assertion (6) states the first subtyping requirement that all methods of `p` must also be provided by `ap`. The assertions (7)-(8) encode the second requirement of the subtyping relation. The relation :> is interpreted in the theory of partial orders. In particular, it is assumed to be reflexive and transitive. Most SMT solvers do not natively support the theory of finite sets and the theory of partial orders, which we use to express the above assertions. However, these theories can be encoded in various ways using more primitive theories that are directly supported. For example, the assertion (6) could simply be expanded into an equivalent conjunction of two assertions consisting of pure equalities:

$$(\mathtt{get} = \mathtt{get} \vee \mathtt{get} = \mathtt{offset} \vee \mathtt{get} = \mathtt{move}) \wedge$$
$$(\mathtt{move} = \mathtt{get} \vee \mathtt{move} = \mathtt{offset} \vee \mathtt{move} = \mathtt{move})$$

The conjunction of the assertions (6)-(15) is unsatisfiable. One possible error source corresponds to dropping assertion (8). This error source tells the programmer that the coercion constraint is invalid. It also identifies which part of the definition of structural subtyping is violated. Another, more useful, error source corresponds to dropping assertion (14). This error source instructs the programmer to change the use of variable `y` on line 14, e.g., by replacing it with `int_of_float y`.

Other Type Systems. There is evidence that our approach can also be applied to a language such as Scala: recent work by Gvero et al. [16] proposes a technique for type-based auto-completion of Scala expressions. This approach relies on a constraint encoding of Scala's type system [10] into first-order logic. Scala supports type inference for declarations in local scopes but uses nominal subtyping. That is, the encoding of the subtyping assertions would be simpler than for OCaml, since one only needs to state the direct subtyping relationships that are explicitly provided by the inheritance tree (plus additional axioms to encode variance of generics). There are other type inference algorithms for specific type systems that are formulated in terms of constraint solving problems [35, 40]. Here, the remaining challenge is to devise an actual SMT encoding of the typing assertions. Support for even more expressive type systems requires additional work. For example, type checking for refinement types as in [34] reduces to checking validity of FOL constraints rather than satisfiability. However, the problem of how to convert such validity queries to satisfiability queries can be addressed by using techniques from [20].

2.5 Interaction with the Programmer

It is not uncommon to utilize the programmer's feedback when localizing type errors [7, 36]. Our approach can be easily extended to support such interactions with the programmer.

Given a compiler provided-ranking criterion and an input program, our approach produces an error source that is minimum with respect to the given criterion, which is then presented to the programmer. What if the programmer wants to disregard the computed error source? The simplest form of interaction is for the programmer to ask for the next best minimum error source with respect to the same ranking criterion. This can be easily supported in our framework, e.g., by adding a blocking clause to the computed typing constraints that rules out the previously computed error source. The solver then computes the next best error source and presents it to the programmer. Many SMT solvers support an incremental mode that preserves the state of the solver across a sequence of satisfiability queries. We can use incremental solving to enumerate the minimum error sources efficiently.

An advantage of our approach is that the underlying algorithm for computing minimum error sources abstracts from the concrete criterion that is used for ranking the error sources. We can therefore support more interesting forms of user interaction where the programmer's feedback is used to modify the ranking criterion on the fly. For example, consider the following scenario which is quite common in practice. Suppose the program consists of two OCaml modules `A` and `B` with `B` depending on `A`. Now suppose that the programmer modifies `B`, introducing a type error that propagates to `A`. The programmer knows that module `B` is still well-typed and that the source of the type error must be located in `A`. Our framework can easily utilize this information to look for the error source in module `A` only. The typing assertions generated from module `B` are simply set as hard, meaning that all the error sources coming from that module should be disregarded, as desired. In fact, we envision a form of interaction where the programmer can tell the system to look for the error source in program parts that are more specific than just modules, e.g., individual functions and let bindings. As program expressions directly map to typing assertions, this form of interaction can be easily supported by adjusting the weights of the corresponding assertions. In general, we believe that the clean separation between the ranking criterion and the constraint solving algorithm that our approach provides enables the systematic study of novel sophisticated type error localization systems.

3. Problem Definition

We now formally define the problem of computing minimum error sources for a given ranking criterion.

Language. For exposition purposes, we base our formal presentation on an idealized language: a lambda calculus with ML-style let polymorphism, named $\lambda^\perp$ for convenience. The syntax of $\lambda^\perp$ is defined as follows:

Expressions	$e ::= x$	variable
	$\mid v$	value
	$\mid e\ e$	application
	$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	let binding
Values	$v ::= n$	integers
	$\mid b$	Booleans
	$\mid \lambda x.\, e$	abstraction
	$\mid \perp$	hole

In addition to the usual constructs of the lambda calculus, our language supports conditional branching and let binding. Specifically, let bindings can be used to define polymorphic functions. Values include integer constants, $n \in \mathbb{Z}$, Boolean constants, $b \in \mathbb{B}$, and lambda abstractions. The special value $\perp$ is called *hole*. We will use holes to define error sources and explain their role in more detail below. Variables x are drawn from an infinite set that is disjoint from all other syntactic constructs. An expression e in which no variables occur free is called a *program*.

We do not define the semantics of $\lambda^\perp$ expressions, as it is irrelevant for our discussion. The reader may assume the expected semantics.

Types. The types of our language are as follows:

$$\begin{array}{rl} \textbf{Monotypes} & \tau ::= \mathsf{bool} \mid \mathsf{int} \mid \alpha \mid \tau \to \tau \\ \textbf{Polytypes} & \sigma ::= \tau \mid \forall \alpha.\sigma \end{array}$$

Monotypes τ include the *base types* bool and int, *type variables* α, which are drawn from an infinite set disjoint from the other types, and *function types* $\tau \to \tau$. A monotype in which no type variables occur is called *ground*. A polytype $\forall\alpha.\sigma$ represents the intersection of all types obtained by instantiating the type variable α in σ by a ground monotype. That is, $\forall\alpha.\sigma$ binds α. We write $\forall\vec{\alpha}.\tau$ as a shorthand for $\forall\alpha_1.\ldots\forall\alpha_n.\tau$ where $\vec{\alpha} = \alpha_1,\ldots,\alpha_n$. We further denote by $fv(\sigma)$ the set of free type variables in type σ. Finally, we write $\sigma[\vec{\beta}/\vec{\alpha}]$ for capture-avoiding substitution of free occurrences of the type variables $\vec{\alpha}$ in σ by the type variables $\vec{\beta}$.

As for other Hindley-Milner type systems, type inference is decidable for $\lambda^\perp$. Expressions therefore do not require explicit type annotations. Typing judgments take the form $\Gamma \vdash e : \tau$. Here, Γ is a *typing environment* that maps variables to types. We write $\emptyset$ for the empty typing environment and extend the function *fv* to typing environments in the expected way. We say that a program p is *well typed* iff there exists a type σ such that $\emptyset \vdash p : \sigma$.

The actual typing rules are shown in Figure 3. The rules are standard, with the exception of the rule [HOLE], which we describe in more detail. The value $\perp$ has the polytype $\forall\alpha.\alpha$. Therefore, the typing rule [HOLE] assigns a fresh unconstrained type variable to each usage of $\perp$. Intuitively, $\perp$ is a placeholder for another program expression. It can be safely used in any context without causing a type error. Changes to hole expressions in an ill-typed program cannot make the program well typed[1].

Minimum Error Sources. Let e be a $\lambda^\perp$ expression. We call a path $\ell \in \mathbb{N}^*$ in the abstract syntax tree representation of e a *location* of e. We denote the set of all locations of e by $Loc(e)$. A location $\ell \in Loc(e)$ uniquely identifies a subexpression of e. We denote this subexpression by $e(\ell)$. Now, let $mask$ be the function that maps an expression e and a location $\ell \in Loc(e)$ to the expression obtained from e by replacing $e(\ell)$ with $\perp$. We extend $mask$ to sets of locations in the expected way.

Definition 1 (Error source). *Let p be a program. A set of locations $L \subseteq Loc(p)$ is an* error source *of p if*

1. *$mask(p, L)$ is well typed*
2. *for all strict subsets L' of L, $mask(p, L')$ is not well typed.*

A ranking criterion allows the compiler to favor error sources of particular interest by assigning appropriate weights to locations. Formally, a *ranking criterion* is a function R that maps a program p to a partial function $R(p) : Loc(p) \rightharpoonup \mathbb{N}_+$. The locations in $Loc(p) \setminus \mathsf{dom}(R(p))$ are considered *hard* locations, i.e., R disregards these locations as causes of type errors. We extend $R(p)$ to a set of locations $L \subseteq Loc(p)$ by defining

$$R(p)(L) = \sum_{\ell \in \mathsf{dom}(R(p)) \cap L} R(p)(\ell) \ .$$

Minimum error sources are error sources that minimize the given ranking criterion.

Definition 2 (Minimum error source). *Let R be a ranking criterion and p a program. An error source $L \subseteq Loc(p)$ is called* minimum error source *of p subject to R if for all other error sources L' of p, $R(p)(L) \leq R(p)(L')$.*

We are interested in the problem of finding a minimum error source for a given program p subject to a given ranking criterion R, respectively, finding all such minimum error sources.

4. Algorithm

In this section, we present our algorithms for computing minimum error sources based on weighted MaxSMT.

4.1 Reduction to Weighted MaxSMT

We first formally define the weighted MaxSMT problem and then explain the actual reduction.

Weighted MaxSMT. The MaxSAT problem takes as input a finite set of propositional clauses $\mathcal{C}$ and finds an assignment that maximizes the number of clauses K that are simultaneously satisfied [26]. MaxSAT can alternatively be viewed as finding the largest subset $\mathcal{C}'$ of clauses $\mathcal{C}$ such that $\mathcal{C}'$ is satisfiable and $\mathcal{C}'$ is a maximum satisfiable subset, $|\mathcal{C}'| = K$. Partial MaxSAT partitions $\mathcal{C}$ into *hard* and *soft* clauses. The hard clauses $\mathcal{C}_H$ are assumed to hold and the goal is to find a maximizing subset $\mathcal{C}'$ of the soft clauses such that $\mathcal{C}' \cup \mathcal{C}_H$ is satisfiable. Weighted Partial MaxSAT, for simplicity referred to as only weighted MaxSAT, adds an integer weight $w_i = w(C_i)$ to each soft clause C_i and asks for a satisfiable subset $\mathcal{C}'$ that maximizes the weighted score:

$$\sum_{C_i \in \mathcal{C}'} w_i \text{ subject to } \mathcal{C}_H \cup \mathcal{C}' \text{ is satisfiable.} \qquad (16)$$

The MaxSMT problem generalizes MaxSAT from working over propositional clauses to a set of assertion formulas $\mathcal{A}$ where each assertion belongs to a fixed first-order theory $\mathcal{T}$. Most concepts directly generalize from MaxSAT to MaxSMT: satisfiability is replaced by Satisfiability Modulo

[1] A similar concept was previously used in [24] where a hole in an OCaml program is represented by an expression that raises an exception.

$$\frac{x : \forall\vec{\alpha}.\tau \in \Gamma \quad \vec{\beta}\ \text{new}}{\Gamma \vdash x : \tau[\vec{\beta}/\vec{\alpha}]}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ [\textsc{App}] \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathsf{bool}}\ [\textsc{Bool}]$$

$$\frac{\Gamma.x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ [\textsc{Abs}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ [\textsc{Cond}] \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathsf{int}}\ [\textsc{Int}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma.x : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \vec{\alpha} = fv(\tau_1) \setminus fv(\Gamma)}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ [\textsc{Let}] \qquad \frac{\alpha\ \text{new}}{\Gamma \vdash \bot : \alpha}\ [\textsc{Hole}]$$

Figure 3. Typing rules for $\lambda^\perp$

Theories [3], partial MaxSMT has hard and soft assertions ($\mathcal{A}_H$ and $\mathcal{A}_S$), weighted partial MaxSMT assigns an integer weight to each soft assertion. We represent weighted MaxSMT instances as tuples $(w, \mathcal{A}_H, \mathcal{A}_S)$ where w is the weight function assigning the weights to the soft assertions.

Theory of Inductive Data Types. Our reduction to the weighted MaxSMT problem generates a typing constraint from the given input program. This constraint is satisfiable iff the input program is well typed. The constraint is then passed to the MaxSMT solver, which computes the minimum error sources. The generated typing constraint hence needs to be expressed in terms of assertions that are interpreted in an appropriate first order theory. We use the theory of inductive data types [4] for this purpose. In this theory, we can define our own inductive data types and then state equality constraints over the terms of that data type. For our encoding, we define an inductive data type Types that represents the set of all ground monotypes of $\lambda^\perp$:

$$t \in \mathsf{Types} ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{fun}(t, t)$$

Here, the term constructor fun is used to encode the ground function types.

The terms in Types are interpreted in the models of the theory of inductive data types. A model of this theory is a first-order structure that interprets the type constructors such that the following axioms are satisfied:

$$\mathsf{int} \neq \mathsf{bool}$$
$$\forall \alpha, \beta \in \mathsf{Types}.\ \mathsf{fun}(\alpha, \beta) \neq \mathsf{int} \land \mathsf{fun}(\alpha, \beta) \neq \mathsf{bool}$$
$$\forall \alpha, \beta, \gamma, \delta \in \mathsf{Types}.\ \mathsf{fun}(\alpha, \beta) = \mathsf{fun}(\gamma, \delta) \Rightarrow \alpha = \gamma \land \beta = \delta$$

That is, the term constructor fun must be interpreted by an injective function, and the interpretation of the terms int and bool is distinct from the interpretation of all other terms. Hence, the axioms exactly encode the equality relation on ground monotypes of $\lambda^\perp$. For the type systems of actual languages such as OCaml, we extend Types with additional type constructors, e.g., to encode user-defined algebraic data types in OCaml programs.

Overview of the Reduction. Next, we describe the actual reduction to weighted MaxSMT. In the following, let p be the program under consideration and let R be the ranking criterion of interest.

At the heart of our reduction is a constraint generation procedure that traverses p and generates a set of assertions in our theory of Types. These assertions encode the typing rules shown in Figure 3. The constraint generation procedure can produce multiple assertions associated with a single location ℓ of p. Suppose R considers a location ℓ to be soft. That is, while searching for minimum error sources the solver must consider two possibilities. If ℓ is in the minimum error source, then none of the assertions generated from the expression $p(\ell)$ need to be satisfied. If on the other hand ℓ is not in the minimum error source, then the type of the expression $p(\ell)$ must be consistent with its context. That is, all the assertions directly associated with ℓ must be satisfied simultaneously. However, some of the assertions generated from subexpressions of $p(\ell)$ may be dropped because the corresponding locations may still be in the minimum error source. Thus, to properly encode the problem of finding minimum error sources, we need to link the assertions of location ℓ together so that the solver considers them as a single logical unit. Also, we need to capture the relationship between the locations according to the structure of p. For this purpose, we associate a propositional variable T_ℓ with every location ℓ. By setting T_ℓ to true, the solver decides that T_ℓ is not in the minimum error source. That is, each assertion associated with a location ℓ_o takes the following form:

$$T_{\ell_n} \Rightarrow \cdots \Rightarrow T_{\ell_1} \Rightarrow T_{\ell_0} \Rightarrow t_1 = t_2 \qquad (17)$$

Here, $\ell_1, \ldots, \ell_n$ are the locations that are reached along the path to ℓ_0 in p (i.e., all proper prefixes of ℓ_0). The terms t_1 and t_2 are terms in our theory of Types. These terms may include logical variables that need to be unified by the MaxSMT solver. Note that the assertion (17) ensures that if any of the T_{ℓ_i} is set to false (i.e., ℓ_i is in the minimum error source), then all the assertions that depend on ℓ_0 are immediately satisfied. In order to simplify the presentation in Section 2, we have omitted the additional propositional variables in our discussion of the examples.

The assertions that encode the typing rules are considered hard. To reduce the problem of finding minimum error sources to weighted MaxSMT, we add to these hard assertions an additional set of clauses, each of which consists of one of the propositional variables T_ℓ. For each ℓ, if $R(p)(\ell)$ is defined, the clause T_ℓ is soft with associated weight $R(p)(\ell)$. Otherwise, T_ℓ is hard. Each solution to this weighted MaxSMT instance then yields a minimum error source by taking all locations ℓ whose clause T_ℓ is not in the solution set.

We now explain these steps in more detail, starting with the generation of the assertions that encode the typing rules.

Assertion Generation. We build on existing work for constrained-based type checking [1, 32, 38] and adapt it for our purposes.

We formalize the assertion generation in terms of a constraint typing relation $\mathcal{A}, \Gamma \vdash p : \alpha$. Here, $\mathcal{A}$ is a set of typing assertions of the form (17). The rules that define the constraint typing relation are given in Figure 4. They are defined recursively on the structure of expressions. To relate the generated typing assertions to p's locations, we assume that every expression is annotated with its location ℓ in p. Note that for a set of assertions $\mathcal{A}$, we write $T_\ell \Rightarrow \mathcal{A}$ to mean $\{\, T_\ell \Rightarrow A \mid A \in \mathcal{A} \,\}$.

We focus on the rules [A-LET] and [A-VAR] as they are the most complex ones. The [A-LET] rule handles let bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ by first computing a set of typing assertions $\mathcal{A}_1$ for e_1. The assertions in $\mathcal{A}_1$ encode all typing assertions imposed by e_1 under the current environment Γ. In particular, they capture the assertions on the type of e_1 itself, which is described by a fresh type variable α_1. To compute the typing assertions $\mathcal{A}_2$ for e_2, the variable x is added to the typing environment Γ. The type of x is described by a typing schema of the form:

$$\forall \vec{\alpha}.(\mathcal{A}_1 \Rightarrow \alpha_1)$$

The typing schema is needed to properly handle polymorphism. It remembers the set of assertions $\mathcal{A}_1$ along with the type variable α_1 that represents the type of e_1 and x. Note that the schema quantifies over all type variables $\vec{\alpha}$ that have been freshly introduced when analyzing e_1 (including α_1). Whenever x is used inside the body e_2 of the let binding, the [A-VAR] rule instantiates $\mathcal{A}_1$ by replacing the type variables $\vec{\alpha}$ with fresh type variables $\vec{\beta}$. The instantiated copy is then added to the other generated assertions. The fresh instances of $\mathcal{A}_1$ ensure that each usage of x in e_2 is consistent with the typing assertions imposed by e_1.

The following lemma states the correctness of the constraint typing relation.

Lemma 1. *Let p be a program, $L \subseteq Loc(p)$, $\mathcal{A}$ a set of typing assertions, and α a type variable such that $\mathcal{A}, \emptyset \vdash p : \alpha$. Then $mask(L, p)$ is well typed iff $\mathcal{A} \cup \{\, T_\ell \mid \ell \notin L \,\}$ is satisfiable in the theory of* Types.

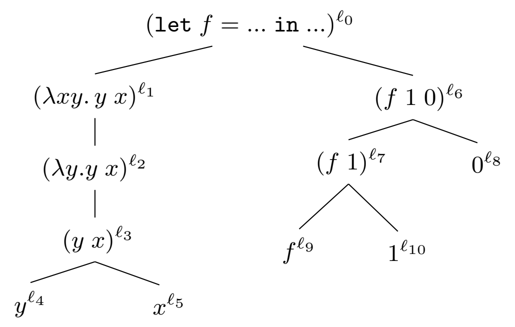

Figure 5. Labeled abstract syntax tree for the program p

The proof of Lemma 1 closely follows that of [38, Lemma 2], modulo reasoning about the auxiliary propositional variables.

Computing Minimum Error Sources. To compute minimum error sources, we generate a weighted MaxSMT instance $I(p, R) = (w, \mathcal{A}_H, \mathcal{A}_S)$ as follows. Let $\mathcal{A}$ be a set of assertions such that $\mathcal{A}, \emptyset \vdash p : \alpha$ for some type variable α. Then define:

$$\mathcal{A}_H = \mathcal{A} \cup \{\, T_\ell \mid \ell \notin \mathsf{dom}(R(p)) \,\}$$
$$\mathcal{A}_S = \{\, T_\ell \mid \ell \in \mathsf{dom}(R(p)) \,\}$$
$$w(T_\ell) = R(p)(\ell), \text{ for all } T_\ell \in \mathcal{A}_S$$

Let $\mathcal{S}$ be a solution of $I(p, R)$. Then define $E(\mathcal{S}) = \{\, \ell \in Loc(p) \mid T_\ell \notin \mathcal{S} \,\}$. The following theorem states the correctness of our reduction.

Theorem 1. *Let p be a program and R a ranking criterion. Then $L \subseteq Loc(p)$ is a minimum error source of p subject to R iff there exists a solution $\mathcal{S}$ of $I(p, R)$ such that $L = E(\mathcal{S})$.*

Example. We conclude the presentation of our algorithm with another example that demonstrates how our approach deals with polymorphic functions. To this end, let p be the following program:

$$\texttt{let}\ f = \lambda x\, y.\ y\ x\ \texttt{in}\ f\ 1\ 0$$

This program is not well typed. The most general type that can be inferred for f from its defining expression is

$$f : \forall \alpha\, \beta.\, \alpha \to (\alpha \to \beta) \to \beta\ .$$

However, the type of 0 is int and not a function type. Hence, applying 0 as second argument to f in the body of the let violates the typing rules.

Figure 5 shows the abstract syntax tree of the program p with each node labeled by its location. Applying the constraint generation rules from Figure 4 then yields the following set of assertions $\mathcal{A}$:

$$\frac{\mathcal{A}, \Gamma.x : \alpha \vdash e : \beta \quad \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \mathsf{fun}(\alpha, \beta)\} \cup \mathcal{A}), \Gamma \vdash (\lambda x.e)^\ell : \gamma} \text{[A-ABS]} \qquad \frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha \quad \mathcal{A}_2, \Gamma \vdash e_2 : \beta \quad \gamma \text{ new}}{T_\ell \Rightarrow (\{\alpha = \mathsf{fun}(\beta, \gamma)\} \cup \mathcal{A}_1 \cup \mathcal{A}_2), \Gamma \vdash (e_1\ e_2)^\ell : \gamma} \text{[A-APP]}$$

$$\frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha \quad \mathcal{A}_2, \Gamma \vdash e_2 : \beta \quad \mathcal{A}_3, \Gamma \vdash e_3 : \gamma \quad \delta \text{ new}}{T_\ell \Rightarrow (\{(T_{\ell_1} \Rightarrow \alpha = \mathsf{bool}), (T_{\ell_2} \Rightarrow \beta = \delta), (T_{\ell_3} \Rightarrow \gamma = \delta)\} \cup \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3), \Gamma \vdash (\texttt{if } e_1^{\ell_1} \texttt{ then } e_2^{\ell_2} \texttt{ else } e_3^{\ell_3})^\ell : \delta} \text{[A-COND]}$$

$$\frac{\alpha \text{ new}}{\emptyset, \Gamma \vdash \bot : \alpha} \text{[A-HOLE]} \qquad \frac{b \in \mathbb{B} \quad \alpha \text{ new}}{\{T_\ell \Rightarrow \alpha = \mathsf{bool}\}, \Gamma \vdash b^\ell : \alpha} \text{[A-BOOL]} \qquad \frac{x : \forall \vec{\alpha}.(\mathcal{A} \Rightarrow \alpha) \in \Gamma \quad \vec{\beta}, \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \mathcal{A}[\vec{\beta}/\vec{\alpha}]), \Gamma \vdash x^\ell : \gamma} \text{[A-VAR]}$$

$$\frac{n \in \mathbb{Z} \quad \alpha \text{ new}}{\{T_\ell \Rightarrow \alpha = \mathsf{int}\}, \Gamma \vdash n^\ell : \alpha} \text{[A-INT]} \qquad \frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha_1 \quad \mathcal{A}_2, \Gamma.x : \forall \vec{\alpha}.(\mathcal{A}_1 \Rightarrow \alpha_1) \vdash e_2 : \alpha_2 \quad \vec{\alpha} = fv(\alpha) \setminus fv(\Gamma) \quad \vec{\beta}, \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \alpha_2\} \cup \mathcal{A}_1[\vec{\beta}/\vec{\alpha}] \cup \mathcal{A}_2), \Gamma \vdash (\texttt{let } x = e_1 \texttt{ in } e_2)^\ell : \gamma} \text{[A-LET]}$$

Figure 4. Rules defining the constraint typing relation for $\lambda^\bot$

$$\begin{aligned}
&T_{\ell_0} \Rightarrow \{\alpha_0 = \alpha_6,\ C_f(\alpha_1'', \alpha_2'', \alpha_3'', \alpha_4'', \alpha_5'', \beta_1'', \beta_2''),\\
&\quad T_{\ell_6} \Rightarrow \{\alpha_7 = \mathsf{fun}(\alpha_8, \alpha_6),\\
&\qquad T_{\ell_7} \Rightarrow \{\alpha_9 = \mathsf{fun}(\alpha_{10}, \alpha_7),\\
&\qquad\quad T_{\ell_8} \Rightarrow \alpha_8 = \mathsf{int},\\
&\qquad\quad T_{\ell_9} \Rightarrow \{\alpha_9 = \alpha_1'',\\
&\qquad\qquad C_f(\alpha_1', \alpha_2', \alpha_3', \alpha_4', \alpha_5', \beta_1', \beta_2')\}\\
&\qquad\quad \},\\
&\qquad T_{\ell_{10}} \Rightarrow \alpha_{10} = \mathsf{int}\}\}
\end{aligned}$$

where

$$\begin{aligned}
&C_f(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \beta_1, \beta_2) \equiv\\
&\quad T_{\ell_1} \Rightarrow \{\ \alpha_1 = \mathsf{fun}(\beta_1, \alpha_2),\\
&\qquad T_{\ell_2} \Rightarrow \{\ \alpha_2 = \mathsf{fun}(\beta_2, \alpha_3),\\
&\qquad\quad T_{\ell_3} \Rightarrow \{\ \alpha_4 = \mathsf{fun}(\alpha_5, \alpha_3),\\
&\qquad\qquad T_{\ell_4} \Rightarrow \alpha_4 = \beta_2,\\
&\qquad\qquad T_{\ell_5} \Rightarrow \alpha_5 = \beta_1\}\}\}
\end{aligned}$$

Note that we introduced a predicate C_f to serve as a shorthand for the instantiated assertions that are associated with the bound variable f.

The assertions in $\mathcal{A}$ are satisfiable in Types if one of the following location variables is assigned false: T_{ℓ_4}, T_{ℓ_8}, T_{ℓ_9}, or any other T_{ℓ_i} where ℓ_i is on the path to one of the locations ℓ_4, ℓ_8, or ℓ_9.

Now consider the ranking criterion R that defines ℓ_9 hard and all other locations ℓ_i soft with rank $R(p)(\ell_i) = n_i$, where n_i is the number of nodes in the subtree of the AST whose root ℓ_i identifies. Solving the weighted MaxSMT instance $I(p, R)$ yields two minimum error sources, each of which contains one of the locations ℓ_4 and ℓ_8.

4.2 Solving Weighted MaxSMT

Finally, we present our algorithm for solving the weighted MaxSMT instances that we generate.

Our algorithm relies on the computation of unsat cores. An unsat core U is an unsatisfiable subset of a set of clauses (respectively, assertions). The maximal satisfying subsets of a set of clauses are closely related to unsat cores [28]. If all strict subsets of U are satisfiable, then U is minimal.

Our algorithm is shown in Figure 6. Is inspired by the approach given in [9]. This approach uses an off-the-shelf partial weighted MaxSAT solver to generate a candidate weighted MaxSAT assignment, called $\mathcal{S}$, on propositional abstractions of $\mathcal{A}$. An SMT solver is used to check whether the candidate assignment $\mathcal{S}$ is theory feasible (line 6). If so, then $\mathcal{S}$ is a weighted MaxSMT answer (line 7). Otherwise, the algorithm learns an unsat core, U, that blocks the candidate (line 8). Let $\neg U$ be a shorthand for the formula representing the negation of the unsat core:

$$\neg U \equiv \bigvee_{a \in U} \neg a \ .$$

The negation of an unsat core is always theory valid. ($\neg U$ is always true in the theory.) This new theory valid lemma is accumulated in a set of lemmas, $\mathcal{L}$, that are passed throughout the execution (line 9). The propositional abstraction of $\mathcal{L}$ is then added as new hard constraints to the weighted MaxSAT solver (line 10). The weighted MaxSAT solver is then asked for a new candidate answer (line 10), and the process repeats until a weighted MaxSMT answer is found (line 11). To ensure the `WMSMT` loop terminates, we begin by checking that $\mathcal{A}_H$ is satisfiable (line 2). To find a solution to the weighted MaxSMT problem, the implementation starts by guessing that all soft assertions can be satisfied $\mathcal{S} = \mathcal{A}_S$ (line 3).

In principle, the procedure `UnsatCore` could return the trivial unsat core, $\mathcal{A}_H \cup \mathcal{L} \cup \mathcal{S}$, and MaxSMT would still be correct and terminating. However, the trivial unsat core is unlikely to be minimal and the MaxSAT solver may end of having to enumerate all subsets of $\mathcal{A}_S$ in this scheme. To accelerate the search, we use a naive implementation of Junker's QuickXplain algorithm [21] for finding a minimal unsat core, U. QuickXplain uses a divide-and-conquer approach to identify assertions that must belong to U using only satisfiability checks. If the sizes of $|U| = k$ and $|\mathcal{A}| = n$, then QuickXplain finds U in $O(k \cdot \log(\frac{n}{k}) + k)$ sat-

```
1  WMaxSMT($w$, $\mathcal{A}_H$,$\mathcal{A}_S$) =
2    if SMT.Solve($\mathcal{A}_H$) = unsat then $\emptyset$
3    else let $\mathcal{S}$,$\mathcal{L}$ = WMSMT($w$,$\mathcal{A}_H$,$\mathcal{A}_S$,[],$\mathcal{A}_S$) in
4         $\mathcal{S}$

5  WMSMT($w$,$\mathcal{A}_H$, $\mathcal{A}_S$, $\mathcal{L}$, $\mathcal{S}$) =
6    if SMT.Solve($\mathcal{A}_H \cup \mathcal{L} \cup \mathcal{S}$) = sat
7    then $\mathcal{S}$, $\mathcal{L}$
8    else let $U$ = UnsatCore($\mathcal{A}_H \cup \mathcal{L} \cup \mathcal{S}$) in
9         let $\mathcal{L}'$ = ($\neg U$)::$\mathcal{L}$ in
10        let $\mathcal{S}'$ = WMaxSAT($w$,$\mathcal{A}_H \cup \mathcal{L}'$,$\mathcal{A}_S$) in
11        WMSMT($w$,$\mathcal{A}_H$,$\mathcal{A}_S$,$\mathcal{L}'$,$\mathcal{S}'$)
```

Figure 6. Weighted MaxSMT using unsat cores

```
1  AllWMaxSMT($w$,$\mathcal{A}_H$, $\mathcal{A}_S$) =
2    let $\mathcal{S}$,$\mathcal{L}$ = WMSMT($w$,$\mathcal{A}_H$,$\mathcal{A}_S$,[],$\mathcal{A}_S$) in
3    AWMSMT($w$,$\mathcal{A}_H$, $\mathcal{A}_S$, $w(\mathcal{S})$, $\mathcal{L}$, [])

4  AWMSMT($w$,$\mathcal{A}_H$, $\mathcal{A}_S$, $W$, $\mathcal{L}$, acc) =
5    let $\mathcal{S}$, $\mathcal{L}'$ = WMSMT($w$,$\mathcal{A}_H$,$\mathcal{A}_S$,$\mathcal{L}$,$\mathcal{A}_S$) in
6    if $w(\mathcal{S})$ < $W$ then acc
7    else let acc' = $\mathcal{S}$::acc in
8         let $\mathcal{A}'_H$ = ($\neg\mathcal{S}$)::$\mathcal{A}_H$ in
9         AWMSMT($w$,$\mathcal{A}'_H$, $\mathcal{A}_S$, $W$, $\mathcal{L}'$, acc')
```

Figure 7. Enumerating all weighted MaxSMT solutions

isfiability checks. Note that as each member of $\mathcal{L}$ is valid, the set $\mathcal{L}$ may safely be dropped in both the `SMT.Solve` (line 6) and `UnsatCore` (line 8) calls.

This procedure is a minor modification of [9] where Lemma Lifting (the `SMT.GetTLemmas` function in [9]) is replaced by `UnsatCore`. Lemma Lifting requires specialized support from the SMT solver, and currently MathSAT is the only SMT solver that explicitly supports this. Our approach uses the SMT solver as a black box, and any SMT solver that supports the theory of inductive data types may be used.

Building upon the MaxSMT code, all solutions to the given MaxSMT instance can be enumerated with repeated calls to MaxSMT and blocking clauses. The `AllMaxSMT` algorithm (Fig. 7) computes a single MaxSMT solution to determine the maximum weight W (line 2). While the algorithm can find a MaxSMT solution $\mathcal{S}$ of weight W, $\mathcal{S}$ is accumulated into `acc`, and is blocked from being a future solution using a new hard constraint (line 7- 9). If the next candidate solution is smaller ($w(\mathcal{S}) < W$), all MaxSMT solutions are in `acc` (line 6).

Preliminary empirical results given in Section 5 show that this "quick-and-dirty" implementation of weighted MaxSMT and unsat core computation is already sufficiently fast to be responsive. A more mature implementation can take advantage of native SMT support for unsat cores.

5. Implementation and Evaluation

We have implemented our algorithm for the Caml subset of the OCaml language and evaluated it on the OCaml benchmark suite from [24]. The tool is available from the following URL: `http://cs.nyu.edu/~zvonimir/minerrloc.tar.gz`.

The vanilla implementation of our algorithm was able to find a minimum error source for 98% of the benchmarks in a reasonable time using a typical ranking criterion. Further, our approach was able to enumerate all minimum error sources on a sizable majority of the benchmarks. While the benchmark programs are quite modest, we find these initial experimental results promising for the overall approach. In addition to our vanilla implementation, we also discuss three optimizations that sacrifice completeness for efficiency by focusing the search to certain parts of the input program.

5.1 Implementation

Our implementation builds on top of the EasyOCaml [14] system, the SMT solver CVC4 [2], and the SAT solver Sat4j [23]. We use EasyOCaml to generate the weighted MaxSMT instances, and a loose combination of CVC4 [2] and Sat4j [23] to solve the generated instances.

Assertion Generation. EasyOCaml implements the type error localization approach described in [17] for a subset of the OCaml language. Similar to our approach, EasyOCaml uses constraint solving for error localization. We tapped into the corresponding code of EasyOCaml and modified it to produce the weighted MaxSMT instances from the input program. The implementation begins by running EasyOCaml on the input program. EasyOCaml produces typing assertions, which we annotate with the locations of the corresponding program expressions in the input source code. These assertions only encode the typing relation. In addition, we output the structure of the input program in terms of its abstract syntax tree edges where the nodes are the locations associated with the typing assertions. This information is sufficient to generate the final typing constraint as described in Section 4.1. In our evaluation, we used a fixed ranking criterion that is defined as follows: (1) all assertions that come from expressions in external libraries are set as hard; (2) all assertions that come from user-provided type annotations are set as hard; and (3) all remaining assertions have a weight equal to the size of the corresponding expression in the abstract syntax tree. We encode the generated assertions in an extension of the `SMT-LIB` 2 language [5] to handle the theory of inductive data types.

Solving the Weighted MaxSMT Instances. Our weighted MaxSMT solver, which we use to solve the generated instances, directly implements the algorithm presented in Section 4.2. The implementation is a thin wrapper of approximately 530 lines of Java code around an SMT solver and a weighted MaxSAT solver. In particular, we use CVC4 to implement the `SMT.Solve` function, and Sat4j to implement the `WMaxSat` function.

Preemptive Cutting. The first optimization of our basic algorithm that we considered is *preemptive cutting*. Here, we run EasyOCaml's inbuilt type checker alongside the constraint generation procedure. As soon as the type checker detects a type error, we stop the constraint generation and analyze only those assertions that have been generated up to this point and in the remainder of the enclosing function definition. This approach reduces the number of generated assertions and thus execution time of the solver. The downside of this approach is that the reported expressions might not constitute a minimum error source for the full program.

Constraint Slicing. Both the basic implementation of our algorithm and the optimization with preemptive cutting take into account all typing assertions that EasyOCaml generates for the considered program portion. We developed an additional optimization that slices the typing constraint by restricting the computed assertions to a *cone of influence* of a known type error location. That is, we first run the OCaml type inference on the input program to obtain a source code location ℓ that is involved in a type error. Next, we compute a subset of the set of typing assertions that we generate for the program as follows. The computed subset is the least fixed point of the function F on sets of typing assertions $\mathcal{A}$, where an assertion a is in $F(\mathcal{A})$ iff either a is directly generated from the expression at location ℓ, or $fv(a) \cap fv(\mathcal{A}) \neq \emptyset$, or a belongs to a set of assertions that is associated with a polymorphic function and that has a variable from one of its instantiations appearing in $fv(\mathcal{A})$. The latter case ensures that if a usage of a polymorphic function affects the type error, then the assertions from all usages of the function as well as the function definition appear in the slice. That way, we still consider other potential type errors caused by the wrong usage of the function.

The rationale behind this optimization is as follows. The input program may be large, but a type error is usually tied to a small part of the program logic. The typing assertions that are not involved in any type error are not needed for finding minimum error sources. Constraint slicing overapproximates the set of relevant typing assertions for a specific type error, weeding out a potentially large number of irrelevant assertions and thereby decreasing the running time of the solver. The downside of this approach is that there are no guarantees that the sources of all type errors are reported, since there can be multiple independent type errors.

5.2 Evaluation

We evaluated our implementation on the OCaml benchmarks used in [24]. The benchmark suite consists of OCaml programs that have been written by students. The programs exhibit various kinds of errors that are statically detected by the compiler. We considered a subset of the original benchmark set consisting of 356 programs. Most of these programs exhibit type mismatch errors, with a few exceptions of programs where a function is called with too many arguments or where the programmer attempted to write to a non-mutable field of a record. The original benchmark suite contains many more programs. However, these other programs do not exhibit type errors but errors that are inherently localized, such as the use of an unbounded value or type constructor. The size of the programs is moderate. The largest program has 397 lines of code, comments included. All of our experiments were conducted on an `Intel(R) Xeon(R)` machine with eight 3.60GHz cores. However, our implementation currently utilizes only a single core.

Quality of Computed Minimum Error Sources. In our first experiment, we assessed how good the produced minimum error sources are at pinpointing the true error source. To this end, we chose 20 programs from the benchmark suite at random, identified the true error source in each program, and compared it against the first minimum error source that was computed by our implementation. As mentioned earlier, we used the ranking criterion that favors error sources of smaller code size and that excludes possible errors sources coming from external functions and type annotations. To identify the true error source we used additional information that was provided by the benchmark suite. Namely, the benchmark suite does not consist of individual programs. Instead, it consists of sequences of modified versions of programs. Each such program sequence was recorded in a single session of interactions between a student and the OCaml compiler. By comparing subsequent program versions within one sequence, we can identify the changes that the student made to fix the reported type errors and thereby infer the true error sources. In the experiment, we used the vanilla implementation without preemptive cutting and constraint slicing.

We classified the result of the comparison as either "hit", "close", or "miss". Here, "hit" means that the computed minimum error source exactly matches the true error source, and "miss" means that there is no relationship between the two. A "close" result means that the locations in the computed minimum error source were close enough to the true error locations so that a programmer could easily understand and fix the type error. For example, if the minimum error source reported the function in a function application instead of the argument of the application (or vice versa), then we considered the result to be close to the true error source.

We then repeated the same experiment but this time using OCaml's type checker instead of our implementation. For each program, we recorded the result of the two experiments. Figure 8 shows the number of obtained result pairs, aggregated over the 20 benchmark programs. As can be seen, on the randomly chosen programs, our approach identifies the true error source more often than the OCaml type checker, even though we were using a rather simplistic ranking criterion. Specifically, our approach missed the true error source in only three programs, whereas OCaml did so in six programs. Despite the subjective nature of true error sources, we believe that this experiment shows the potential of our approach to providing helpful error reports to the user.

Assertion generation. The next experiment we conducted was measuring the time spent on generating the typing constraints. For the versions without constraint slicing, this includes the time spent on generating typing assertions in EasyOCaml and the time taken by our post-processing procedure to convert EasyOCaml's output into the final SMT constraints. Recall that in the version with preemptive cutting, we stop the constraint generation as soon as EasyOCaml detects a type error. When slicing is used, we additionally include the time taken by OCaml's type checker as well as the time taken to compute the slice using the information provided by OCaml's error report. Figure 9 shows the correlation between the measured time and the number of produced assertions for all four versions of our implementation. As it can be seen, the overall time needed for the constraint generation is reasonably small. This is to be expected as the input programs are of moderate size. Note, however, that the additional constraint slicing step does not significantly increase the constraint generation time.

It can be seen from Figure 9 that the number of assertions gets large for some programs. Figure 10 provides a more detailed correlation between program size and the number of generated assertions. In general, the number of generated assertions gets bigger as programs do. The number of assertions depends on the number of usages of polymorphic functions since each use of a polymorphic function results in copying the associated set of assertions. This explains the difference in the number of generated assertions for programs of similar size. Also, this explains why the number of generated assertions for some programs is large even if we use the cutting and slicing optimizations.

Computing minimum error sources. The last experiment we performed had the goal of measuring the time spent computing the minimum error sources. We broke the benchmark suite of 356 programs into 8 groups according to their size in the number of lines of code. The first group includes programs consisting of 0 and 50 lines of code, the second group includes programs of size 50 to 100, and so on as shown in the first column of Figure 11. The numbers in parenthesis indicate the number of programs in each group.

In the first experiment, we measured the time spent computing a single minimum error source for the whole input programs. We expect the computation of a single error source to be the main use case in practice. Compilers in general might want to avoid burdening the programmer with multiple error sources and only show a single one instead. Also, the purpose of ranking criteria is to favor a few specific error sources of high quality, rather than getting a large number of possible candidates.

AST size criterion	OCaml	# of outcomes
hit	miss	3
hit	close	2
hit	hit	4
close	miss	1
close	close	6
close	hit	1
miss	miss	2
miss	close	1
miss	hit	0

Figure 8. Quality of AST ranking criterion compared to OCaml's type checker in pinpointing the true error source in 20 benchmark programs

The results of the experiment are shown in Figure 11. The table shows statistics about the execution time, the weight of the minimum error sources, as well as the number of assertions passed to the solver. Note that the number of these assertions is smaller than the number of generated assertions. This is because we conjoined all assertions that are generated for each location to a single assertion. The main reason for doing so is that when a variable appears in a small number of assertions, the solver needs to do less internal handling of such variables, thus gaining better performance. For all three measured values, the minimum, median, and maximum values are given for each family. As it can be seen, the execution times are reasonably small. However, for some programs the framework spends a couple of minutes computing the solution. The slowdown in our implementation mostly occurs in the cases where a huge number of constraints has been generated.

In some cases we observed that QuickXplain [21] unsat core computation caused a significant slowdown. This algorithm makes only black-box satisfiability queries, not relying on specialized support from the solver. We expect specialized techniques to make fewer satisfiability queries. Additionally, we observed that the QuickXplain algorithm is sensitive to the ordering of the assertions. For instance, the order where the assertions for the location variables came before the actual typing assertions performed better than the order where typing assertions came first. The former is the order we use in our implementation.

In addition to computing a single minimum error source, we conducted a separate experiment where we computed all minimum error sources. We report the number of minimum error sources as well as the total computation time. Not surprisingly, computing all error sources takes more time. We observed that in this experiment the running time for computing all error sources appears to grow sublinear with the number of error sources. We hypothesize that our algorithm takes advantage of the capabilities of incremental solvers. That is, the algorithm in Figure 7 for computing all MaxSMT solutions remembers the set of computed lemmas

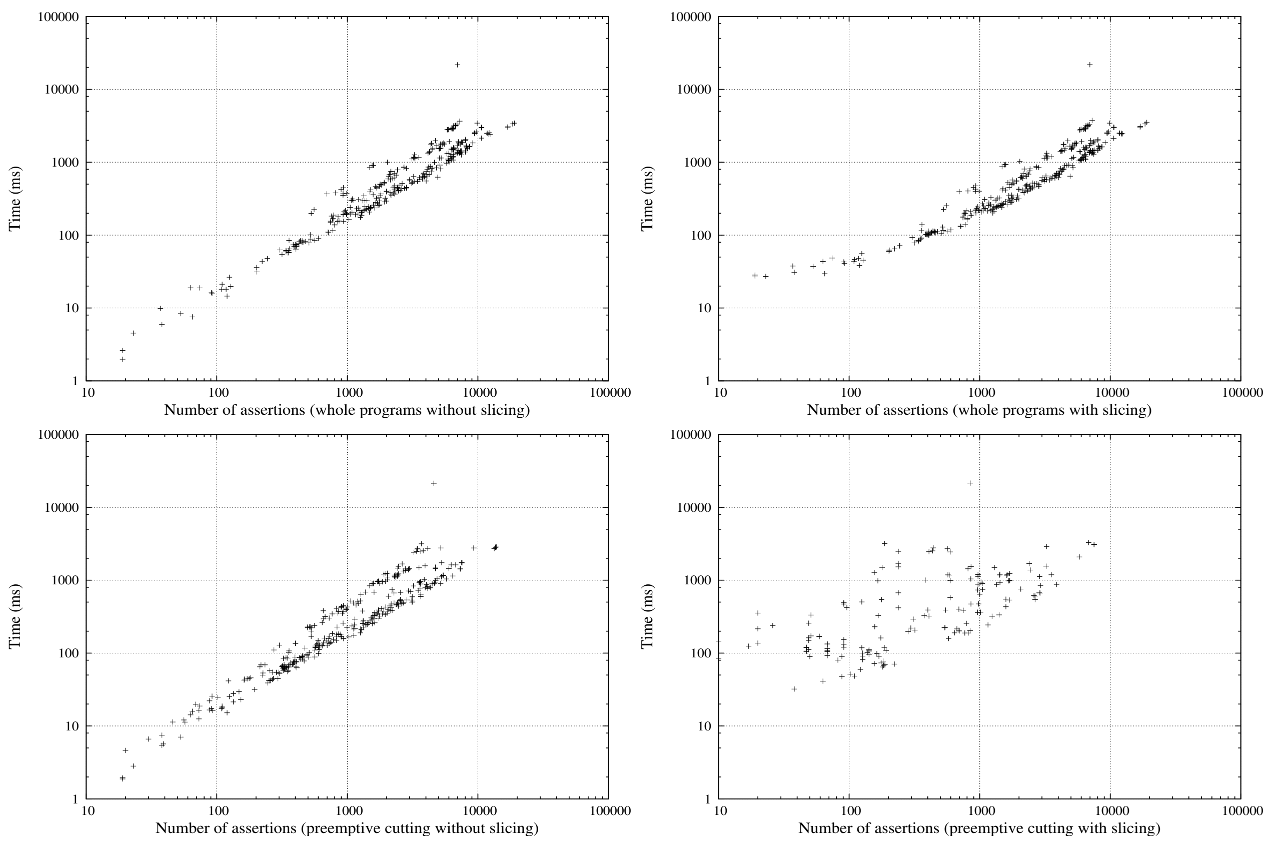

Figure 9. Constraint generation execution time correlated with constraint size

across subsequent calls to the MaxSMT solver, reducing the running times of these calls. However, computing all minimum error sources can be significantly more expensive than computing a single error source if a program has multiple independent type errors. In this case, the solver finds all minimum error sources for each individual type error and then enumerates all possible combinations between those. This means that the number of minimum error sources may grow exponentially with the number of independent errors. In our future work, we plan to address this issue by isolating error sources one from another.

We repeated the same experiments using the slicing optimization. The results of these experiments are shown in Figure 12. The slicing optimization performs better, as expected. Likewise, we repeated the experiments with preemptive cutting (Figure 13) and with both preemptive cutting and constraint slicing (Figure 14). The cutting optimization further improves the running times. The execution times for the version with preemptive cutting are comparable to those achieved with the approach due to Zhang and Myers [42], who used the same benchmarks in their evaluation and also use the cutting optimization.

Further Performance Improvements. There are a number of opportunities for further improving the performance of our implementation and to reduce the overall execution time. The implementation of our MaxSMT solver is quite simplistic. We plan to improve our MaxSMT solver by taking advantage of more mature algorithms for computing unsat cores and by incorporating advanced techniques such as lemma lifting. However, the main bottleneck of our current implementation is that the set of generated assertions can grow very large. As can be seen in Figure 10, the number of generated assertions grows exponentially with the program size, even if we use the cutting and slicing optimizations. We next explain in more detail the source of this exponential explosion and discuss how we intend to address it.

6. Taming the Exponential Explosion

The constraint typing relation that is at the heart of our reduction suffers from one major drawback: the size of the generated assertion set can grow exponentially in the size of the program. The reason for this behavior is the interplay between the rules [A-LET] and [A-VAR] in Fig. 4. These rules govern the handling of let-bound polymorphic variables: for each use of a let-bound variable x in the body of the let expression, the rule [A-VAR] adds a fresh instance of the typing constraint that is associated with x by rule [A-LET] to the set of assertions. Hence, the number of generated assertions grows exponentially with the nesting depth of let bindings in the program.

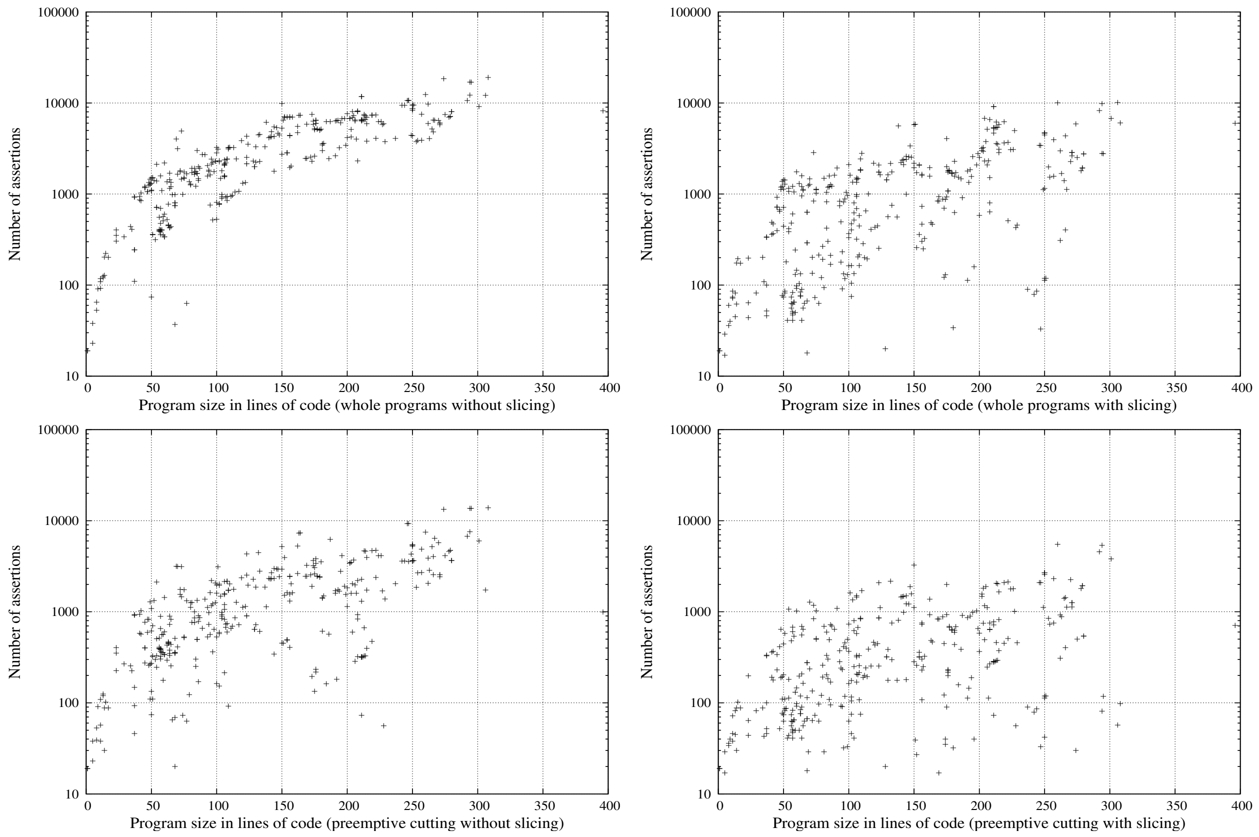

Figure 10. Constraint size correlated with program size

The exponential worst-case complexity of the constraint generation cannot be avoided in general, as the type checking problem for Hindley-Milner type systems is EXPTIME-complete [22, 27]. This complexity carries over to the problem of finding minimum error sources. Nevertheless, actual implementations of type checkers for OCaml and other languages that are based on the Hindley-Milner type system achieve good performance in practice. This raises the question whether one can achieve comparable performance with our approach by avoiding unnecessary instantiation of assertions during constraint generation. We propose two possible solutions to this problem that we will explore in future work.

Lazy Quantifier-Based Instantiation. Our first solution exploits that all instantiated copies of a set of assertions share the same structure. We can thus introduce an auxiliary predicate symbol as a shorthand for the assertion set. The auxiliary predicate symbol is defined using a universally quantified axiom. This axiom is passed to the MaxSMT solver as an additional hard assertion. Whenever we need to instantiate the assertion set associated with a let-bound variable in the [A-VAR] rule, we instead instantiate the predicate associated with that variable. We have already used this idea implicitly in the example at the end of Section 4.1, where we introduced the predicate C_f to represent the assertion set that describes the type of the polymorphic function f. The improved constraint typing relation is obtained by modifying the rule [A-LET] as shown in Figure 15.

When the MaxSMT solver solves the assertions, it will generate the copies of the assertion sets lazily by instantiating the appropriate axioms. Note that the defining axioms of the auxiliary predicates satisfy stratification restrictions that ensure the underlying satisfiability problem remains decidable. That is, the defining axioms of the auxiliary predicates may mutually depend on each other, but these dependencies are not cyclic. In the worst case, the SMT solver will trigger as many axiom instantiations as there are instances generated by the naive constraint typing rules.

The advantage of this solution is that it can be easily implemented by using an SMT solver that provides a quantifier instantiation engine. Also, we only need a single MaxSMT query to compute the minimum error sources. The disadvantage is that the solver may still consider many unnecessary instantiations during its search.

Lazy Unification-Based Instantiation. In actual compilers, the exponential blow-up of type inference is tamed by computing the *principal type* of the defining expression of each let-bound variable x. The principal type of an expression is a type such that all other types for this expression are an instance of the principal type. For Hindley-Milner type systems, principal types correspond to *most general*

Group	Constraints			Single minimum error source						All minimum error sources					
				Error source weight			Time (sec)			# of sources			Time (sec)		
	min	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**
0-50 (47)	15	220	524	1	1	10	0.06	0.29	8.86	1	2	12	0.06	0.33	10.95
50-100 (102)	33	519	1130	1	1	14	0.06	0.71	10.02	1	2	20	0.08	0.84	11.23
100-150 (65)	520	968	1805	1	1	81	0.26	1.75	82.27	1	2	87	0.27	2.31	83.77
150-200 (57)	1109	1504	2002	1	1	12	0.49	4.66	24.53	1	1	19	0.59	4.79	26.38
200-250 (53)	1067	1732	2091	1	1	17	0.92	6.50	58.80	1	2	10	1.08	8.25	121.97
250-300 (28)	1497	2401	2911	1	1	24	0.84	4.79	153.50	1	2	4	0.98	5.72	218.73
300-350 (3)	2277	2445	3273	2	4	4	9.68	16.71	86.71	4	12	16	10.91	37.75	93.63
350-400 (1)	2657	2657	2657	1	1	1	11.06	11.06	11.06	8	8	8	23.76	23.76	23.76

Figure 11. Statistics for framework execution for whole programs without constraint slicing

Group	Constraints			Single minimum error source						All minimum error sources					
				Error source weight			Time (sec)			# of sources			Time (sec)		
	min	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**
0-50 (47)	15	125	473	1	1	8	0.06	0.21	8.17	1	2	12	0.06	0.24	10.99
50-100 (102)	18	261	1067	1	1	14	0.06	0.40	9.87	1	2	20	0.06	0.49	11.11
100-150 (65)	15	616	1787	1	1	81	0.05	1.35	81.82	1	2	87	0.05	1.39	82.61
150-200 (57)	106	819	1871	1	1	12	0.08	1.98	20.45	1	1	19	0.12	2.58	29.87
200-250 (53)	28	1230	1855	1	1	17	0.05	5.54	40.15	1	1	10	0.06	5.50	62.92
250-300 (28)	96	1501	2778	1	1	24	0.11	3.24	108.14	1	2	4	0.12	3.68	114.25
300-350 (3)	1617	1753	2503	2	2	4	8.71	9.76	80.16	4	4	12	9.78	24.15	85.83
350-400 (1)	2415	2415	2415	1	1	1	10.05	10.05	10.05	8	8	8	22.24	22.24	22.24

Figure 12. Statistics for framework execution for whole programs with constraint slicing

Group	Constraints			Single minimum error source						All minimum error sources					
				Error source weight			Time (sec)			# of sources			Time (sec)		
	min	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**
0-50 (47)	15	177	518	1	1	6	0.06	0.13	1.65	1	2	10	0.06	0.16	3.57
50-100 (102)	18	341	986	1	1	8	0.07	0.34	6.13	1	1	8	0.06	0.37	10.01
100-150 (65)	85	755	1595	1	1	81	0.12	0.94	26.61	1	2	29	0.15	1.21	36.15
150-200 (57)	106	1030	1841	1	1	12	0.09	0.92	10.95	1	1	19	0.08	1.22	26.14
200-250 (53)	52	1138	2042	1	1	17	0.08	0.96	23.23	1	1	10	0.08	1.12	28.00
250-300 (28)	1023	1734	2896	1	1	24	0.58	2.45	49.35	1	2	4	0.61	3.39	61.26
300-350 (3)	1040	2193	3203	1	2	4	0.84	8.39	9.96	1	4	16	0.99	8.25	21.42
350-400 (1)	762	762	762	1	1	1	0.94	0.94	0.94	8	8	8	2.13	2.13	2.13

Figure 13. Statistics for framework execution for preemptive cutting without constraint slicing

Group	Constraints			Single minimum error source						All minimum error sources					
				Error source weight			Time (sec)			# of sources			Time (sec)		
	min	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**
0-50 (47)	9	70	406	1	1	6	0.05	0.10	1.04	1	2	10	0.05	0.12	3.43
50-100 (102)	18	120	816	1	1	8	0.05	0.16	3.81	1	1	8	0.05	0.18	9.82
100-150 (65)	15	320	1592	1	1	81	0.05	0.45	27.04	1	2	29	0.05	0.56	34.06
150-200 (57)	13	287	1763	1	1	12	0.05	0.33	21.46	1	1	19	0.05	0.34	22.75
200-250 (53)	28	501	1826	1	1	17	0.06	0.61	10.49	1	1	10	0.06	0.62	12.77
250-300 (28)	23	1206	2369	1	1	24	0.05	1.64	48.56	1	2	4	0.06	1.97	51.53
300-350 (3)	55	516	1599	1	2	2	0.09	0.33	8.95	1	4	4	0.11	0.34	12.47
350-400 (1)	630	630	630	1	1	1	0.85	0.85	0.85	8	8	8	1.81	1.81	1.81

Figure 14. Statistics for framework execution with preemptive cutting and constraint slicing

unifiers (mgu) of the generated typing constraints and can be computed efficiently using Robinson's unification algorithm [33]. The principal type of x can then be substituted in at each place where x is used in the body of the let expression. This eliminates the repeated analysis of x's typing constraints for each usage of x.

A first attempt to apply this idea to the problem of computing minimum error sources is to replace the instantiated typing assertions of a bound variable's defining expression by the entailed mgu. This approach would yield an equisatisfiable typing constraint and avoid the exponential blow-up of the constraint size in practice. Unfortunately, this approach

$$\frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha_1 \quad \mathcal{A}_2, \Gamma.x : \forall\vec{\alpha}.(\{C_x(\vec{\alpha})\} \Rightarrow \alpha_1) \vdash e_2 : \alpha_2 \quad \vec{\alpha} = fv(\alpha) - fv(\Gamma) \quad \vec{\beta}, \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \alpha_2\} \cup \{\forall\vec{\alpha}.\, C_x(\vec{\alpha}) \Rightarrow \mathcal{A}_1\} \cup C_x(\vec{\beta}) \cup \mathcal{A}_2), \Gamma \vdash (\texttt{let } x = e_1 \texttt{ in } e_2)^\ell : \gamma} \text{[A-LET]}$$

Figure 15. Modified rule [A-LET] that avoids an exponential explosion in the size of the generated assertion set

also potentially eliminates minimum error sources from the solution set: the minimum error source of a type error that becomes manifest in the use of a let-bound variable may be in the defining expression of that variable. However, we can use most general unifiers to conservatively approximate the instantiated typing assertions in the body of each let expression. To ensure that minimum error sources are not lost, we guarantee that the weight of the mgu is smaller or equal to the weight of any propositional variable in the approximated assertions. This approximation is then iteratively refined by instantiating the constraints for let-bound variables lazily, guided by the computed minimum error sources, until a fixpoint is reached.

We describe this algorithm in more detail. During constraint generation, for each let-bound variable x defined at some location ℓ_x, unify the generated constraints $\mathcal{A}_x$ for x. If unification fails, proceed as in the basic algorithm, i.e, instantiate $\mathcal{A}_x$ at all locations where x is used. If unification succeeds, let $\forall\vec{\alpha}.\tau$ be x's mgu. Then, at each location ℓ where x is used, add a constraint

$$A_\ell \equiv T_\ell \Rightarrow T_{\ell_x} \Rightarrow \alpha = \tau[\vec{\beta}/\vec{\alpha}]$$

where α is the type imposed by the context of ℓ_i and $\vec{\beta}$ are fresh type variables. Assign to T_{ℓ_x} the minimum weight of all locations that the defining expression of x recursively depends on.

Solve the current weighted MaxSMT instance for a solution $\mathcal{S}$. For every let-bound variable x, if $T_{\ell_x} \notin \mathcal{S}$, then instantiate $\mathcal{A}_x$ by replacing the constraint A_ℓ with

$$A_\ell \equiv T_\ell \Rightarrow T_{\ell_x} \Rightarrow \mathcal{A}_x[\vec{\beta}'/\vec{\alpha}']$$

for each ℓ where x is used. Here $\vec{\alpha}'$ are the free variables of $\mathcal{A}_x$ that are unconstrained by the environment and $\vec{\beta}'$ are fresh. Set the weight of T_{ℓ_x} back to its original value. This process is repeating until a solution S is found such that all of the locations not included in $\mathcal{S}$ are fully instantiated. This algorithm avoids the instantiation of constraints for let-bound variables that are not involved in the type error.

In practice, even in large programs only few let-bound variables actually contribute to a type error. This suggests that the algorithm outlined above should perform well. We plan to implement this algorithm in our future work.

7. Conclusion

We have presented a general framework for localization of type errors based on constraint solving. What makes our approach unique is its support for ranking criteria that enable compilers to control which error sources should be preferred in the search. Our algorithm then guarantees to find minimum error sources subject to the given ranking criterion. By reducing the problem of finding minimum error sources to weighted MaxSMT, the algorithm can be easily adapted to support rich type systems. We reported on results obtained for type error localization in OCaml programs using a naive implementation of our basic algorithm. Our experiments suggest that the approach is effective at identifying minimum error sources and has the potential to significantly improve the quality of type error reports.

Acknowledgments

This work was supported by the National Science Foundation under grant CCS-1350574. We would like to thank Danfeng Zhang from Cornell University for helping us understand the EasyOCaml software.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41. ACM, 1993.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177. Springer-Verlag, 2011.

[3] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [6], February 2009.

[4] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.

[5] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard – version 2.0. In *SMT*, 2010.

[6] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[7] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL*, pages 583–594. ACM, 2014.

[8] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP*, ICFP '01, pages 193–204. ACM, 2001.

[9] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. A Modular Approach to MaxSAT Modulo Theories. In *SAT*, pages 150–165, 2013.

[10] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A Core Calculus for Scala Type Checking. In *MFCS*, volume 4162 of *LNCS*, pages 1–23. Springer, 2006.

[11] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.

[12] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1995.

[13] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[14] EasyOCaml. `http://easyocaml.forge.ocamlcore.org`. [Online; accessed 10-March-2014].

[15] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2005.

[16] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38. ACM, 2013.

[17] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, pages 189–224, 2004.

[18] The Haskell Programming Language. `http://www.haskell.org/`. [Online; accessed 15-March-2014].

[19] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:2960, 1969.

[20] M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV*, pages 504–509. Springer-Verlag, 2011.

[21] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.

[22] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML Typability is DEXTIME-Complete. In *CAAP*, pages 206–220, 1990.

[23] D. Le Berre, A. Parrain, et al. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[24] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI*. ACM Press, 2007.

[25] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. OCaml Manual: Module Pervasives. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`. [Online; accessed 14-March-2014].

[26] C. M. Li and F. Manyà. *MaxSAT, Hard and Soft Constraints*, chapter 19, pages 613–631. Volume 185 of Biere et al. [6], February 2009.

[27] H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *POPL*, pages 382–401, 1990.

[28] J. Marques-Sila and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In K. Gulati, editor, *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 171–182. Springer New York, 2011.

[29] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[30] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP*, pages 15–26. ACM Press, 2003.

[31] OCaml. `http://ocaml.org`. [Online; accessed 2-February-2014].

[32] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[33] J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.

[34] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, pages 159–169. ACM, 2008.

[35] T. Schrijvers, S. L. P. Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *ICFP*, pages 341–352. ACM, 2009.

[36] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Haskell*, pages 72–83. ACM, 2003.

[37] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91. ACM, 2004.

[38] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner style type systems in constraint form. *Res. Rep. ACRC-99-009, University of South Australia, School of Computer and Information Science. July*, 1999.

[39] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, pages 5–55, 2001.

[40] D. Vytiniotis, S. L. P. Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

[41] M. Wand. Finding the source of type errors. In *POPL*, pages 38–43. ACM, 1986.

[42] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *POPL*, pages 569–581. ACM, 2014.