A Practical Framework for Type Inference Error Explanation

Calvin Loncaric
University of Washington
Seattle, WA, USA
loncaric@cs.washington.edu

Satish Chandra Cole Schlesinger
Manu Sridharan
Samsung Research America
Mountain View, CA, USA
{schandra,cole.s,m.sridharan}@samsung.com

Abstract

Many languages have support for automatic type inference. But when inference fails, the reported error messages can be unhelpful, highlighting a code location far from the source of the problem. Several lines of work have emerged proposing error reports derived from *correcting sets*: a set of program points that, when fixed, produce a well-typed program. Unfortunately, these approaches are tightly tied to specific languages; targeting a new language requires encoding a type inference algorithm for the language in a custom constraint system specific to the error reporting tool.

We show how to produce correcting set-based error reports by leveraging *existing* type inference implementations, easing the burden of adoption and, as type inference algorithms tend to be efficient in practice, producing error reports of comparable quality to similar error reporting tools orders of magnitude faster. Many type inference algorithms are already formulated as dual phases of type constraint generation and solving; rather than (re)implementing type inference in an error explanation tool, we isolate the solving phase and treat it as an oracle for solving typing constraints. Given any set of typing constraints, error explanation proceeds by iteratively removing conflicting constraints from the initial constraint set until discovering a subset on which the solver succeeds; the constraints removed form a correcting set. Our approach is agnostic to the semantics of any particular language or type system, instead leveraging the existing type inference engine to give meaning to constraints.

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

Keywords Type Error Diagnosis, Type Inference

1. Introduction

Type inference is often sold as a boon to programmers, offering all the benefits of static typing without the overhead of type annotations. It is increasingly popular, with modern languages like Scala supporting local type inference and older languages like C++ adding type inference features such as the `auto` keyword. Reporting insightful type inference errors in the presence of such language features is difficult because the problem is ambiguous. To illustrate, Chen and Erwig [4] give the example of the OCaml expression "`not 1`," which fails to typecheck because `not` requires a boolean input. Is the error in the use of `not` or the use of `1` or the existence of the entire expression? This question cannot be answered without knowing the programmer's original intent, and inferring that intent is an unsolved—and possibly unsolvable—problem.

The expression "`not 1`" concisely pinpoints this ambiguity, but the uses and definitions that give rise to typing conflicts in real programs often appear much farther apart. Adding features to the language and type system also adds complexity in how different program points contribute to failure in type inference, making insightful error reporting even more difficult.

Modern compilers—such as the OCaml compiler—take the pragmatic approach of reporting errors eagerly, as soon as a typing conflict is discovered. This is easy to implement but often misses the actual location of the error. For example, consider the following code snippet, which Zhang and Myers [26] drew from a body of student assignments [17] and simplified for clarity:

```
1  let rec loop lst acc =
2    if lst = [] then
3      acc
4    else
5      print_string "foo"
6    in
7  List.rev (loop [...] [(0.0, 0.0)])
```

There are two noteworthy constraints: The types of "`acc`" and "`print_string "foo"`" on lines 3 and 5 must be the same, and the type of "`acc`" must be the same as the type of `[(0.0, 0.0)]`, which is used as an argument to `loop` on line 7. Without line 7, the first constraint forces `acc` to have

type `unit`, the return type of `print_string`. After inferring this, the OCaml compiler reaches line 7 and reports that `loop` is invoked with `[(0.0, 0.0)]` instead of the unit value. The actual error, as reported and fixed by the student in question, was the use of `print_string` on line 5.

Type inference algorithms often comprise two phases: One generates a set of constraints relating the types of different program points, and another solves the constraints to generate appropriate types [1, 18, 25]. A line of work has emerged developing error explanation engines based on specialized type constraint languages and solvers [11, 20, 21, 26, 27]. For example, recent work proposes encoding type inference as an SMT problem. By translating typing constraints to SMT constraints, an off-the-shelf solver can replace the custom solver, which leads to an elegant error reporting mechanism: On failure, a MaxSMT solver can produce a *correcting set* containing constraints that, when fixed, results in a well-typed program. By considering *every* constraint violation, rather than just the first encountered, this approach can better pinpoint the root cause of type errors [20, 21]. Another similar approach is to produce correcting sets based on a Bayesian interpretation of the program [26, 27]. However, there are two limitations in these kinds of approaches to error reporting—ease of adoption and scalability:

- For a compiler to take advantage of these error reporting tools, one must (re)implement type inference in a different constraint system—a non-trivial task. For example, finding an efficient SMT encoding is still an open question for many type inference systems, even those naturally formulated as type constraint generation and solving. Even when an encoding can be found, reimplementing type inference represents substantial redundant effort.
- In our experience, specialized type inference algorithms remain more efficient than the constraint solvers used to build error reporting engines. Relying on a MaxSMT solver limits scalability as the size of programs—and the number of constraints—increases. The encoding problem itself can also impact performance, as some type system features, such as the parametric polymorphism found in OCaml, require an exponential number of SMT constraints compared to lines of code.[1]

To address these limitations, we present MYCROFT,[2] a framework that enables compiler writers to augment *existing* (constraint-based) type inference implementations to produce correcting sets, rather than reimplementing type inference in a distinct, specialized constraint system. The key technical insight lies in decoupling the constraint generation and solving phases of an existing type inference implementation and using the type solver as an oracle to decide whether a collection of typing constraints is satisfiable. By leveraging the existing type inference implementation, MYCROFT is agnostic to the language and type system.

This approach also improves performance. By using the existing type solver, MYCROFT works with the existing constraint system and avoids inefficiencies in encoding one constraint system in another. (In particular, we avoid the constraint explosion that arises from encoding parametric polymorphism, as in [20, 26].) Moreover, the type solver and its constraint language can be optimized with respect to domain specific knowledge, which can lead to better performance than an off-the-shelf SMT solver. Finally, we experiment with selecting candidate correcting sets using a greedy approximation algorithm rather than an optimal exponential-time one, which leads to further improvements.

To use MYCROFT, a compiler writer must make two changes to type inference: Factor out constraint generation from solving, and augment solving to produce unsat cores. We ease adoption by developing an API for instrumenting a type solver to record the constraints that influence each type variable assignment, and from that generate an unsat core on failure. Using this instrumentation, we implement error explanation for OCaml and for SJSx [3], a type system that enables ahead-of-time compilation for a large subset of JavaScript.[3] The former allows us to compare performance with prior work, while the latter demonstrates that adoption is not hampered by complex type system features.

We evaluate MYCROFT by comparing its performance and the quality of its error reports against two competing tools, MinErrLoc [20] and SHErrLoc [26], on a benchmark of student homework assignments drawn from the Seminal project [17]. Our results show that MYCROFT produces error explanations of comparable quality with substantial performance improvement. We also report on our experience porting unrestricted JavaScript programs to the SJSx fragment.

Contributions. In summary, our contributions are:

- MYCROFT: A framework for enabling adoption of correcting set-based error explanation by retrofitting constraint-based type inference algorithms. (Section 3.)
- NP-hard and greedy approximation algorithms for selecting the best candidate correcting sets. (Section 3.3.)
- Two case studies—one for OCaml and one for SJSx—that show how to extract unsat cores from the solvers. (Section 4.)
- An evaluation on a subset of the Seminal benchmark suite [17] demonstrating that the greedy approach produces error messages of comparable quality to previous work at substantially lower run-time cost. (Section 7.)

[1] The MinErrLoc tool overcomes this with clever heuristics to approximate principal types in their SMT encoding of OCaml type inference [21], but the problem remains for other encodings.

[2] MYCROFT is named for Mycroft Holmes, the brother of Sherlock.

[3] We refer to the work of Chandra *et al*. [3] as SJSx to distinguish it from SJS [5].

2. Our Approach

MYCROFT takes as input two components that, together, comprise a type inference implementation:

- A *type constraint generator* that produces a set of constraints for a given program and associates those constraints with program points, and
- A *type solver* that produces either a type assignment satisfying the constraints or, on failure, a core of unsatisfiable constraints.

Notably, MYCROFT is agnostic to the semantics of constraints or the program from which they came. Given a program, it proceeds as follows.

1. Generate an initial set of typing constraints using the type constraint generator.
2. Submit the constraint set to the type solver.
3. (a) On success, stop.
   (b) On failure, partition the original constraints into a candidate correcting set and typing set, and submit the typing set to the type solver.

Step 3 repeats until sufficient constraints have been moved to the correcting set for the solver to succeed. As an example, consider the following code sample drawn from [20].

```
1  let f x = print_int x in
2  let g x = x + 1 in
3  let x = "hi" in
4    f x;
5    g x
```

The functions `f` and `g` use their argument as an integer, but the value supplied at the call sites is a string. The two library functions `print_int` and `(+)` are ascribed types $int \to unit$ and $int \to int \to int$ in the context. For brevity, let us assume the let-bound variables `f`, `g`, and `x` are initially ascribed fresh unification variables $F_{in} \to F_{out}$, $G_{in} \to G_{out}$, and X. The code then gives rise to the following set of constraints.

$$F_{in} = int \quad (1)$$
$$F_{out} = unit \quad (2)$$
$$G_{in} = int \quad (3)$$
$$G_{out} = int \quad (4)$$
$$X = string \quad (5)$$
$$F_{in} = X \quad (6)$$
$$G_{in} = X \quad (7)$$

Constraints (1, 2) are derived from the first line, corresponding to the application of `print_int` to `x` and the return type of `print_int x`. Constraints (3, 4) are similar, and the remaining constraints capture that `x` is a string (5) and an argument to `f` and `g` (6, 7). There are two unsat cores in this set of constraints—$\{1, 5, 6\}$ and $\{3, 5, 7\}$—which reflect the type mismatch between the definition of `x` and its use as an argument to `f` and `g`, whose parameters are used as integers.

We have already seen the first step of MYCROFT's algorithm (generating constraints). Suppose we submit those constraints to the type solver (Step 2) and walk through four iterations of Step 3:

- Step 3(b). The solver fails and produces a (potentially non-minimal) unsat core. Suppose it is $\{1, 2, 5, 6\}$, which is indeed non-minimal—constraint (2) is unnecessary. As this is the only unsat core discovered so far, MYCROFT selects one constraint from the unsat core, hoping to break the conflict. To demonstrate the impact of non-minimal unsat cores, suppose we select the extraneous constraint (2).
- Step 3(b). In the next round, MYCROFT invokes the solver again, withholding constraint (2). This time, suppose the solver returns $\{1, 5, 6\}$, which happens to be minimal, as constraint (2) is not available this round. The candidate selection algorithm then selects the smallest set of constraints that overlaps with both unsat cores—an easy task in this case, as the latter is a subset of the former. Suppose it selects (1).
- Step 3(b). With the constraint $F_{in} = int$ removed, the solver fails and produces the remaining unsat core, $\{3, 5, 7\}$. The algorithm selects the set $\{5\}$ as the smallest set that intersects with all three unsat cores generated so far.
- Step 3(a). The solver succeeds with the correcting set $\{5\}$ and typing set $\{1, 2, 3, 4, 6, 7\}$.

Hence, MYCROFT produces the singleton set containing $X = string$ as the smallest correcting set, suggesting that the value bound to `x` be changed to an integer type. As we see from the first round, the algorithm tolerates non-minimal unsat cores at the cost of additional rounds, because removing an extraneous constraint does not break the underlying conflict. Section 3.4 proves that MYCROFT terminates and produces a minimal correcting set, even in the presence of non-minimal unsat cores.

2.1 Constructing Human-Readable Error Reports

One advantage that correcting sets offer over other error reporting mechanisms is that a great deal of information is available after solving to construct a readable error report. Specifically, for every broken constraint, the tool has access to: (1) the constraint itself, (2) the original program point that produced the constraint, and (3) a type assignment that arose after all broken constraints were removed.

MYCROFT is equipped with a default pretty printer that converts this information into a human-readable error message. In the example above, MYCROFT produces the correcting set $\{X = string\}$ as well as the typing that results from removing that constraint, where `x`—in order to satisfy the remaining constraints—is typed at int. In this case, the human-readable report reads "`"hi"` on line 3 is a value of type `string` but should be an expression of type `int`."

Compiler writers may also supply an optional, custom pretty printer to MYCROFT, which is a function producing a human-readable error report from a correcting set, the origin of the constraints therein, and the types computed for those points after the correcting set was removed.

2.2 Targeting More Expressive Type Systems

Because MYCROFT is agnostic to the meaning of constraints, it can easily be retargeted to complex type systems, so long as their type inference algorithms can be expressed as type constraint generation and solving. Our work on SJSx serves as an example; SJSx supports mutable records, prototype inheritance, and subtyping [3]. We have used MYCROFT to implement type error explanation for SJSx. Because of the richness of the SJSx type system, inference is more complex than for Hindley-Milner languages like OCaml. Useful error messages from the type inference engine are important here as well. We present the details of SJSx as well as how we augmented the SJSx type solver for MYCROFT in Section 4.3.

2.3 Relationship to Prior Work

MYCROFT is spiritually akin to MinErrLoc [20], which proposes using weighted maximum satisfiability modulo theories (MaxSMT) to produce minimal correcting sets. Indeed, we had initially intended to adopt this approach directly for SJSx, but we were stymied by two difficulties. First, it was not obvious how to reduce type inference for the SJSx type system to SMT—and doing so would require abandoning the substantial work that went into developing the type inference algorithm in the first place. Second, we found that the MaxSMT approach does not scale to the size of programs we anticipated.

SHErrLoc [26] also represents a notable point in this space: SHErrLoc reports potential error locations ranked using Bayesian techniques based on the assumption that the programmer's code is mostly correct. SHErrLoc requires reducing type inference to its custom constraint language, which we found, as with SMT, to be a daunting task for SJSx.

3. Architecture of MYCROFT

Figure 1 presents the high-level MYCROFT algorithm. Given a program p, MYCROFT uses the compiler-writer supplied constraint generator to extract typing constraints from the program, which are then passed to the type solver. On failure, the explanation engine selects a candidate correcting set $\mathcal{F}'$, removes the correcting set from the constraint set, and submits the resulting subset to the type solver. This cycle—managed by the recursive function `FIND_FIX`—continues until the explanation engine produces constraints on which the type solver succeeds.

3.1 The Type Constraint Generator

The type constraint generator—`TYPECGEN.Generate()` on line 2 in Figure 1—analyzes the syntax of a program and produces

```
1  MYCROFT(p) =
2    let C = TYPECGEN.Generate(p)
3    return FIND_FIX(C, [], ∅)
4
5  FIND_FIX(C_in, L, F) =
6    let C = C_in - F
7    if TYPESOLVER.Solve(C) = sat
8    then return F
9    else let U = TYPESOLVER.UnsatCore()
10        let L' = U::L
11        let F' = FindCandSet(C_in, L')
12        return FIND_FIX(C_in, L', F')
```

Figure 1. The MYCROFT algorithm.

a set of constraints that, if satisfied, lead to a valid typing for the program. It is supplied to MYCROFT by the compiler writer. The constraints can be simple or complex, driven by the needs of the underlying type system—MYCROFT is agnostic to the meaning of the constraints or the program itself.

3.2 The Type Solver

The compiler-writer supplied type solver decides whether a set of constraints leads to a valid typing. It is invoked as `TYPESOLVER.Solve()` on line 7 of Figure 1. Although MYCROFT can make use of a type solver that simply accepts or rejects sets of constraints, it dramatically improves performance if the type solver also reports *why* a given set of constraints failed in the form of an unsat core.

Lines 9–11 show where the unsat core is generated (by `TYPESOLVER.UnsatCore()`) and used: An unsat core U is appended to the list of cores $\mathcal{L}$ generated so far and passed to `FindCandSet()` to find the next candidate correcting set. Finally, line 12 recursively invokes `FIND_FIX` with the latest candidate correcting set and unsat cores.

3.3 Finding a Correcting Set

Given a program that produces a set of constraints $\mathcal{C}$, the algorithm in Figure 1 solves a combinatorial optimization problem by searching for a set of constraints $\mathcal{F}$ such that

$$\underset{\mathcal{F} \subseteq \mathcal{C} \wedge sat(\mathcal{C}-\mathcal{F})}{\arg\min} \; |\, \mathcal{F} \,| \tag{8}$$

where sat stands for the type solver and determines whether a set of constraints is satisfiable. In short, it seeks a minimal correcting set $\mathcal{F}$ such that $\mathcal{C} - \mathcal{F}$ is satisfiable. Even though there may be many minimal solutions, we have found that selecting one arbitrarily works well in practice (Section 7). The core of the algorithm hinges on selecting a candidate correcting set (the call to `FindCandSet` on line 11); each round (*i.e.* each invocation of `FIND_FIX`) evaluates the best correcting set found thus far. On failure, the function `FindCandSet()` uses the most recent unsat core and all the others generated thus

```
1  FindCandSet(C, L) =
2    if L is empty
3    then return ∅
4    else find c ∈ C maximizing
5            count([ l | l ∈ L, c ∈ l ])
6         let L' = [ l | l ∈ L, c ∉ l ]
7         return {c} ∪ FindCandSet(C, L')
```

Figure 2. Greedy solution to the hitting set problem.

far to produce a new candidate correcting set. The new set includes at least one constraint from each unsat core.

Selecting a candidate correcting set is an instance of the *hitting set* problem, which is a specialization of the *set cover* problem.

Definition 1 (Hitting set)**.** *Given a finite set $\mathcal{X}$ and a family $\mathcal{L}$ of non-empty subsets of $\mathcal{X}$, a set $\mathcal{H} \subseteq \mathcal{X}$ is a* hitting set *if it has a non-empty intersection with each $\mathcal{S} \in \mathcal{L}$. The* hitting set problem *is the task of finding a minimal-size hitting set given some $\mathcal{X}$ and $\mathcal{L}$.*

Appendix A shows that the hitting set problem is NP-hard by demonstrating the relationship to the set cover problem.

Implementing `FindCandSet()` is a matter of solving the hitting set problem with $\mathcal{X}$ instantiated with the constraints $\mathcal{C}$ and $\mathcal{L}$ with the set of unsat cores. MYCROFT includes two implementations of `FindCandSet()`: an optimal, exponential time implementation based on a reduction to MaxSAT, and an approximate, polynomial time implementation.

MaxSAT. MYCROFT's optimal `FindCandSet()` implementation uses a Partial MaxSAT solver. A Partial MaxSAT solver takes as input a set of *hard* boolean constraints (which must be satisfied in the solution) and *soft* boolean constraints (which may or may not not be satisfied in the solution) and produces the largest set of soft boolean constraints that are mutually satisfiable with the set of hard constraints.

MYCROFT reduces the hitting set problem on type constraints to Partial MaxSAT as follows. First, each type constraint is assigned a unique boolean variable, and each of these boolean variables gets asserted as a soft boolean constraint for the Partial MaxSAT solver. Next, each unsat core becomes a hard boolean constraint stating that at least one boolean variable from among its type constraints must be false. Given this input, the Partial MaxSAT solver will produce the largest set of soft boolean constraints such that at least one boolean variable from each unsat core is false. The complement of this set corresponds to a candidate correcting set—*i.e.* a minimal set of type constraints such that every unsat core is covered. This approach was also taken by previous work [20].

Greedy Set Cover. The set cover problem has a known greedy approximation algorithm that yields a cover within a factor of $\Theta(\log\ n)$ of the optimal, where n is the size of the universe of elements [6]. In the case of MYCROFT, the universe of elements is the set of constraints appearing in any unsat core returned by the solver (which may be much smaller than the total set of constraints produced by the constraint generator).

The greedy approximation algorithm for set cover gives rise to a greedy approximation algorithm for the hitting set problem. Figure 2 shows MYCROFT's greedy implementation of `FindCandSet()`. At each iteration, `FindCandSet()` finds the constraint `c` that appears in ("hits") the greatest number of unsat cores in $\mathcal{L}$. Those cores are removed from $\mathcal{L}$, and the algorithm repeats until $\mathcal{L}$ is empty.

Implementing `FindCandSet()` with an approximation algorithm removes a performance bottleneck within each round of `FIND_FIX`, but at the cost of precision: An imprecise correcting set may contain more constraints than necessary. Since each constraint in the correcting set corresponds to an error message for the user, an imprecise correcting set causes the tool to report more errors than actually exist in the program source. Fortunately, we found this problem to be small in practice (Section 7.4).

We finish this subsection by comparing MYCROFT with MinErrLoc [20], which has had a strong influence on our work. In fact, both these techniques can be seen as instantiations of the algorithm presented in Figure 1.

- Where MinErrLoc invokes an SMT solver, MYCROFT instead calls out to the type solver to determine satisfiability and extract unsat cores (lines 7 and 9 in Figure 1).
- Where MinErrLoc relies on MaxSAT to find the next candidate correcting set each round, MYCROFT instead uses `FindCandSet()` (line 11), which can be implemented by any algorithm that solves the hitting set problem. MYCROFT includes two implementations of this procedure: an optimal strategy using a conversion to MaxSAT, and the greedy approximation shown in Figure 2.

3.4 Properties of MYCROFT

MYCROFT is guaranteed to terminate and to produce a minimal correcting set (or $\Theta(\log\ n)$ approximation), provided that certain properties hold of the type constraint generator and solver supplied by the compiler writer.

Termination. As part of the compilation tool chain, it is critical that MYCROFT terminates on all inputs. MYCROFT relies on an easily-established property of the type constraint solver to ensure termination: The solver cannot introduce new constraints. Rather, it must return a subset of the original constraints on failure. The termination argument hinges on the fact that each round produces a unique candidate correcting set not seen in previous rounds.

Lemma 1 (Candidate correcting sets are unique)**.** *If round n (*i.e. *the nth invocation of `FIND_FIX`) produces a candidate correcting set $\mathcal{F}_n$, then there does not exist a round $0 < k < n$ where $\mathcal{F}_k = \mathcal{F}_n$.*

Proof. The proof goes by induction on n, with $n = 1$ as the (trivial) base case. Consider the inductive case, wherein the nth round checks if $\mathcal{F}_{n-1}$ is a correcting set, and if that fails, produces a candidate correcting set $\mathcal{F}_n$. If $\mathcal{F}_{n-1}$ is not a correcting set, then the solver will produce an unsat core $\mathcal{U}_n$. By design, $\mathcal{F}_n$ will have a non-empty intersection with each $\mathcal{U}_i$, $0 < i \leq n$. However, $\mathcal{F}_{i-1} \cap \mathcal{U}_i = \emptyset$ for all $0 < i \leq n$, because $\mathcal{U}_i$ is a subset of $\mathcal{C}_{in} - \mathcal{F}_{i-1}$, which are the constraints submitted to the solver in the i'th round and those exclude the correcting set that the previous round produces (by definition, $\mathcal{F}_0 = \phi$). Therefore, $\mathcal{F}_n \neq \mathcal{F}_{i-1}$ for all $0 < i \leq n$.

□

Termination follows from Lemma 1.

Theorem 1 (MYCROFT terminates)**.** *Given a program p,* MYCROFT *terminates and produces a correcting set.*

Proof. Each round (*i.e.* each invocation of `FIND_FIX`) either succeeds (and terminates) or fails and selects a new, unique subset of constraints as a candidate correcting set. There are finitely many such subsets, and so the algorithm will eventually try them all. The subset containing every constraint will succeed, because removing all constraints necessarily removes all conflicting constraints. Hence, MYCROFT terminates and returns the candidate correcting set produced on the final round.

□

Note that the constraints the solver returns need not form a minimal unsat core, and in fact, they need not form an unsat core at all. Returning any subset of the constraints at each iteration will ensure termination. However, non-minimal unsat cores will degrade the efficiency of the algorithm. To illustrate, suppose an unsat core contains a constraint c that, when removed, does not break the conflict that generated the core. If c is selected for the candidate correcting set, then the conflict is not resolved, and the solver will return another unsat core containing the same conflict (but without the extraneous constraint c that was just removed). Hence, overly-large unsat cores will incur additional iterations of `FIND_FIX`. In the extreme case, if the constraint solver trivially returns the entire set of constraints as the unsat core, then the `FIND_FIX` procedure will degrade to iteratively exploring all subsets of size 1, then size 2, and so on until a correcting set is found.

Minimality. Producing minimal correcting sets excludes unrelated program points from the error explanation. MYCROFT relies on a single property of the type solver to guarantee minimality: The solver must return a *valid* unsat core that does, in fact, contain constraints that are unsatisfiable, although the unsat core need not itself be minimal.

Theorem 2 (MYCROFT produces a minimal correcting set)**.** *If* MYCROFT *produces a correcting set $\mathcal{F}$ for a program p, then there does not exist a smaller correcting set for p.*

Proof. Suppose a smaller correcting set $\mathcal{F}$' exists. Let $\mathcal{L}$ be the set of unsat cores that MYCROFT used in its final round to produce $\mathcal{F}$. By the validity assumption we make of the solver, it follows that every unsat core $u \in \mathcal{L}$ contains a conflict; hence, to be a correcting set, $\mathcal{F}$' must hit every unsat core u. But this is a contradiction: MYCROFT, by Definition 1, produces a minimal hitting set $\mathcal{F}$ for $\mathcal{L}$, but there exists a smaller hitting set $\mathcal{F}$'.

□

A similar argument shows that MYCROFT produces a correcting set within a factor of $\Theta(\log\ n)$ when a greedy approximation of the hitting set algorithm is used, such as the one in Figure 2.

4. Extracting Unsat Cores

We have deployed MYCROFT as an error explanation engine for two existing languages: OCaml and SJSx. OCaml has been the subject of prior error explanation research, allowing us to compare with other work (see Section 7). SJSx is a subset of JavaScript designed to enable aggressive static optimizations. It is equipped with a type system that admits mutable records, subtyping, and prototype inheritance. Targeting SJSx illustrates how MYCROFT can integrate with a complex type inference implementation.

Section 4.1 presents a general approach for augmenting an arbitrary type constraint solver to produce unsat cores. Sections 4.2 and 4.3 show how we specialize this approach for the OCaml and SJSx type constraint solvers, respectively.

4.1 A General Approach to Unsat Core Tracking

There is a generic way to augment any type constraint solver to produce an unsat core: The QuickXplain [13] algorithm produces unsat cores from arbitrary black-box solvers. When the set of input constraints is unsatisfiable, QuickXplain iteratively minimizes the set of constraints until it cannot be reduced further without becoming satisfiable. This unsatisfiable, irreducible set of constraints is a small unsat core. However, there is a high cost to find this unsat core: QuickXplain may make many calls to the underlying solver.

For many type constraint solvers, the overhead of making many calls to the solver can be avoided by augmenting the solver itself to track which constraints contribute to the conflict. At a high level, constraint solvers explore subsets of constraints, using them to compute an assignment to variables within the constraints. Hence, we can produce an unsat core by tracking:

- the constraints under consideration at any given time, and
- which constraints influence each type assignment.

```
class CoreTracker<Constraint, Type>

  def push(cs : Set<Constraint>)          : Void
  def pop()                               : Void

  def recordCurrentConstraints(v : Type) : Void
  def retrieveConstraints(v : Type)
      : Set<Constraint>

  def getCore()                       : UnsatCore
```

Figure 3. Unsat core tracking API.

As an example, consider again the following constraints generated from the example in Section 2:

$$F_{in} = int \quad (1)$$
$$F_{out} = unit \quad (2)$$
$$G_{in} = int \quad (3)$$
$$G_{out} = int \quad (4)$$
$$X = string \quad (5)$$
$$F_{in} = X \quad (6)$$
$$G_{in} = X \quad (7)$$

The solver begins by considering $F_{in} = int$ and assigns int to F_{in}, and so marks F_{in} as influenced by constraint (1). Next, the solver considers $F_{out} = unit$, forgetting for the moment the first constraint. The second constraint induces the assignment of $unit$ to F_{out}, and the solver marks constraint (2) as influencing F_{out}.

Each constraint is so treated, until the solver reaches constraint (6). Here, the solver observes a conflict: X has been unified with $string$ but F_{in} has been unified with int. The solver has recorded that constraint (1) is responsible for the former unification and constraint (5) for the latter. Together with constraint (6), these constraints form a small unsat core.

Figure 3 shows an API that captures the operations necessary to instrument a type solver, simplifying the task of tracking this information. Figures 4c and 5 in the following sections give examples of its use in extracting unsat cores from OCaml and SJSx. In this API, the `push` method is used to indicate that a set of constraints is now under consideration (active), and `pop` removes from consideration the most recently pushed set. The `recordCurrentConstraints` method is called whenever the assignment for a type changes. When `recordCurrentConstraints` is called, the `CoreTracker` will remember the set of active constraints as affecting the given type. The `retrieveConstraints` method returns all constraints that have so far affected a given type, which is used when the current assignment to one type causes a change in the assignment of another. Finally, the `getCore` method returns an unsat core consisting of all constraints that have been `pushed`.

Tracking a constraint set for every type variable can be expensive. For efficiency, we use *derivation trees* parameterized by constraint types to efficiently implement the sets (`Set<Constraint>`) used in `CoreTracker`. A derivation tree is either empty, contains a single constraint, or is a union of two or more derivation trees. Using this representation, each $\cup$ operation on constraint sets takes constant time, thus minimizing overhead. Extracting the unsat core after solving requires walking over the tree to collect all constraints at the leaves. This approach offers a good balance of memory usage to run-time overhead.

The `CoreTracker` class is agnostic to the nature of constraints and types, making it applicable to arbitrary type constraint systems and solvers.

4.2 Unsat Cores from OCaml

As a proof of concept, we have developed a simple implementation of the OCaml type inference algorithm, in the style of Algorithm $\mathcal{W}$ [7] with a separation of constraint generation and solving akin to Rémy's formalization [23]. We begin by presenting a fragment of the OCaml language and show how to augment type inference to produce unsat cores. Figure 4 shows our constraint language, generation rules, and solver pseudocode for OCaml type inference. While we present only a small subset of the OCaml language here (Figure 4a), our implementation supports additional OCaml features such as pattern matching, data type declarations, and references. MYCROFT does not yet support records, although we believe the extension to be straightforward.

Type Constraint Generation. Figure 4b outlines constraint generation, which we implemented by instrumenting the OCaml compiler. Notably, the constraint language ranges over type equalities, which includes generalized polymorphic types (Poly(τ)). As a result, the constraint generator only generates a single constraint for each occurrence of a polymorphic function. Other constraint systems instead copy the constraints associated with the definition and bind them to fresh unification variables for each occurrence, leading to an exponentially increased number of constraints [11]. Our use of explicit polymorphic constraints is similar to the notion of *instantiation constraints* seen in some Hindley-Milner style constraint systems [22].

Although this implementation generates relatively fewer constraints, type solving still has worst-case exponential time [15]. However, the lazy instantiation of Poly(τ) yields a large benefit: In practice, the number of constraints related to a type is much larger than the type itself. Thus, constraint duplication is expensive but lazy instantiation is cheap. Improved performance behavior is one benefit of using a custom constraint system rather than converting the problem into an off-the-shelf format, and it is discussed further in Section 7.2.

Type Solving. Figure 4c shows in pseudocode the unification-based solver for our constraint system. The algorithm is standard, but our handling of Poly(τ) terms deserves some attention. Whenever a Poly(τ) term is encountered, a fresh copy of τ's current assignment is created. This mimics the instantiation procedure from Algorithm $\mathcal{W}$ [7], which instantiates a type with a fresh copy whenever it appears on the

$$
\begin{aligned}
e &::= x \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
&\quad \mid e_1\ e_2 \mid \lambda x.e \mid \text{let rec } x = e_1 \text{ in } e_2 \\
\tau &::= \alpha \mid \text{Int} \mid \tau \to \tau \mid \text{Poly}(\tau) \\
c &::= \tau_1 = \tau_2
\end{aligned}
$$

(a) Core language containing expressions e, type terms τ, and type constraints c.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow \emptyset} \qquad \frac{e \in \mathbf{Z}}{\Gamma \vdash e : \text{Int} \rightsquigarrow \emptyset}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \chi_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \chi_2 \quad \text{fresh}(i) \quad \text{fresh}(o)}{\Gamma \vdash e_1\ e_2 : o \rightsquigarrow \chi_1 \wedge \chi_2 \wedge \tau_1 = (i \to o) \wedge \tau_2 = i}$$

$$\frac{\text{fresh}(t) \quad \text{fresh}(v) \quad \Gamma, x : t \vdash e : \tau \rightsquigarrow \chi}{\Gamma \vdash \lambda x.e : v \rightsquigarrow \chi \wedge v = (t \to \tau)}$$

$$\frac{\text{fresh}(t) \quad \Gamma, x : t \vdash e_1 : \tau_1 \rightsquigarrow \chi_1 \quad \Gamma, x : \text{Poly}(t) \vdash e_2 : \tau_2 \rightsquigarrow \chi_2}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau_2 \rightsquigarrow t = \tau_1 \wedge \chi_1 \wedge \chi_2}$$

(b) Constraint generation, written $\Gamma \vdash e : \tau \rightsquigarrow \chi$, produces a type τ and list of constraints χ for a given term e in context Γ.

```
def solve(cs):
  a = { } # maps variable→type assignment
  for c in cs:
    match c with:
      (τ1 = τ2) →
        push(new Set(c)); unify(τ1, τ2, a); pop()

def unify(τ1, τ2, a):
  match τ1, τ2 with:
    Poly(σ), _ → unify(fresh(σ, a), τ2, a)
    _, Poly(_) → unify(τ2, τ1, a)
    α, _ →
      if a[α]:
        push(retrieveConstraints(α))
        unify(a[α], τ2, a)
        pop()
      else:
        a[α] = τ2
        recordCurrentConstraints(α)
    _, α → unify(τ2, τ1, a)
    Int, Int → pass
    (i1 → o1), (i2 → o2) →
      unify(i1, i2, a)
      unify(o1, o2, a)
    _, _ → raise UnificationFailure( getCore() )
```

(c) Constraint solving.

Figure 4. Type constraint generation and solving for a subset of OCaml. Highlighted program points show where we extend standard unification with derivation tracking to produce unsat cores.

right-hand side of a `let`-expression. In order to perform this transformation safely, all constraints generated from the body of a polymorphic function (*i.e.* constraints on τ) must appear *before* all polymorphic constraints generated from uses of the polymorphic function (*i.e.* constraints on Poly(τ)), ensuring the solver computes an assignment for τ before instantiating uses of Poly(τ). Our constraint generation rules are organized to ensure this. Note that Algorithm $\mathcal{W}$ implicitly enforces the same order of constraint unification by invoking unification during a structural traversal of the program syntax that explores the left-hand side of `let` statements before the right.

Unsat Core Generation. Calls to the unsat core tracking API (Figure 3) have been highlighted in Figure 4c. Each constraint in the constraint system is visited exactly once in the `solve` procedure, and is `push`ed while the implied unification is resolved. Whenever a type variable gets a new value it is marked using `recordCurrentConstraints`, and whenever an already-assigned variable is visited all the contributing constraints are also `push`ed. Thus when `getCore` is called at a failure point, all constraints that ever contributed to the conflict will be returned.

We can see the interaction between the solver and unsat core tracking API in more detail by revisiting the constraint system from Section 2. For the sake of illustration, suppose the solver first visits constraints (1), (5), and (6):

$$cs = \{F_{in} = int,\ X = string,\ F_{in} = X\}$$

The solver begins by selecting $F_{in} = int$ from cs. Invoking `push(new Set(`$F_{in} = int$`))` marks that the solver is currently considering this constraint, followed by `unify(`F_{in}`, `int`, a)`. The call to `unify()` matches the $(\alpha, \_)$ case, and, as `a[`F_{in}`]` is undefined, it takes the false branch of the if statement. The solver assigns `a[`F_{in}`] = `int, and then

`push(recordCurrentConstraints(`F_{in}`))`

marks that the assignment to F_{in} was influenced by the current constraints. After returning from `unify()`, the solver invokes `pop()` to remove the constraint from the set under consideration, making way for the next iteration of the loop to consider the next constraint. Processing the constraint $X = string$ proceeds similarly.

After processing $F_{in} = int$ and $X = string$, `a = {`F_{in}`:`int`, `X`:`$string$`}`. Furthermore, the `CoreTracker` knows that constraint (1) affects the type of F_{in} and constraint (5) affects the type of X.

Finally the solver visits constraint $F_{in} = X$. As before, `push(new Set(`$F_{in} = X$`))` marks this constraint as under consideration and then `unify(`F_{in}`, `X`, a)` is called. Since

F_{in} is a type variable and `a[`F_{in}`]` is defined, the solver takes the true branch of the type variable case. Invoking `push(retrieveConstraints(`F_{in}`))` adds constraint (1) to the set of constraints under consideration, and the solver recursively invokes `unify(`int`, `X`, a)`.

The `(_, `α`)` case reverses the arguments, invoking `unify(`X`, `int`, a)`, which again leads to the true branch of the `(`α`, _)` case: Invoking `push(retrieveConstraints(`X`))` adds constraint (5) to the working set of constraints.

After substituting for X, the solver recursively invokes `unify(`$string$`, `int`, a)`. This causes a `UnificationFailure` exception, and `getCore()` returns the constraints currently under consideration, which, in this case, includes $F_{in} = X$ (the constraint currently selected in the loop in `solve()`), $F_{in} = int$ (the constraint that resolved F_{in}), and $X = string$ (the constraint that resolved X). These constraints form an unsat core.

4.3 Unsat Cores from SJSx

SJSx is a typed subset of JavaScript that enables aggressive static code optimization. Based on the work of Choi *et al.* [5], SJSx admits mutable records, width subtyping, prototype inheritance, and recursive types. This section briefly outlines the terms, types, and constraints that comprise SJSx, as well as a high-level overview of the type solving algorithm extended with constraint tracking to produce unsat cores. The goal of this section is to illustrate how MYCROFT integrates with a complex type system without a deep understanding of the semantics of terms or constraints. Nevertheless, readers interested in a full account of the SJSx type system and inference algorithm can consult the published report by Chandra *et al.* [3].

Type Constraint Generation. The SJSx language ranges over integers, let bindings, variable use and declaration, objects (*i.e.* records with mutable fields and prototype inheritance), field projection and assignment, and method attachment and invocation. Much of the complexity in the SJSx type system deals with careful tracking of the fields of objects—whether they reside locally within an object, in its prototype chain, or not at all—and whether the fields its methods use are present locally. Rows, object base types and type qualifiers, and type variables all help define the constraint system, which is composed of two parts: the constraints themselves, and rejection criteria, which are a syntactically distinct form of constraint.

The SJSx type system is a black box to MYCROFT; the constraint generator emits conjunctions of constraints and acceptance criteria defined in [3], and MYCROFT in turn submits subsets of these to the solver. As part of the SJSx implementation, we extended the constraint generation algorithm to emit hard and soft constraints. Hard constraints capture "obvious" facts, such as "a variable `x` has the same type everywhere." Soft constraints reflect constraints the programmer can affect, such as field access or variable assignment. We found this extension to be straightforward.

```
def solve(cs):
  # normalize
  replace every τ1 = τ2 in cs with τ1 <: τ2 ∧ τ2 <: τ1

  # initialize
  lb = { } # maps variable→lower bound
  ub = { } # maps variable→upper bound
  for each τ in terms(cs):
    lb[τ] = ∅
    ub[τ] = ∅

  # solve
  until fixpoint, for each c in cs:
    match c with:
      τ1 <: τ2 →
        push(new Set(c) ∪
             retrieveConstraints(τ1) ∪
             retrieveConstraints(τ2))
        lb[τ2] = lb[τ1] ∪ lb[τ2]
        if lb[τ2] changed:
          recordCurrentConstraints(τ2)
        ub[τ1] = ub[τ1] ∪ ub[τ2]
        if ub[τ1] changed:
          recordCurrentConstraints(τ1)
        pop()
      ...

  # check
  for τ in terms(cs):
    push(⋃σ∈τ retrieveConstraints(σ))
    glb = greatestLowerBound(ub[τ])
    for T in lb[τ]:
      if T ≮: glb:
        raise TypeError( getCore() )
    pop()
```

Figure 5. Pseudocode implementation of the SJSx type solver. The solver maintains upper and lower bounds for each type variable and propagates information from the type constraints at each iteration. Highlighted code shows where we augment the solver to produce unsat cores.

Type Solving. Figure 5 shows the SJSx type inference implementation in pseudocode. At a high level, the solver associates upper and lower bounds, represented as sets of types, with each variable in the program; each constraint constrains these bounds. As the subtyping relation defines a lattice, the solver iterates—propagating bounds information—until it reaches a fixed point. Critically, constraints guide the flow of information from the bounds of one variable to another, which presents an opportunity to track the effects of constraints.

The pseudocode in Figure 5 is intended to illustrate the essence of unsat core instrumentation. For a more thorough

treatment of the SJSx inference algorithm, we encourage the reader to consult [3].

Unsat Core Generation. The highlighted lines in Figure 5 show the additions required to track unsat cores. As with the OCaml solver, we `push` the constraints that affect each type variable, beginning with the constraint `c` selected each round, as well as any constraints involved in resolving either type in the constraint. Whenever the assignment for a type's upper or lower bound changes, it is marked by a call to `recordCurrentConstraints`. After the bounds have stabilized, if any lower bound type is not a subtype of the greatest lower bound (in the subtype lattice) of the corresponding upper bound, then a type error exists. Invoking `getCore()` produces an unsat core with the constraints that contributed to establishing the upper and lower bounds of τ. The unsat cores generated by the instrumented SJSx type solver are not guaranteed to be minimal. However, we found them to be small in practice.

A Note on Reducing SJSx Type Inference to SMT. Unsat core generation could conceivably be achieved in SJSx by converting the constraints to an SMT formula and using the unsat core extraction functionality of an off-the-shelf SMT solver. Unfortunately, several features of the SJSx type system make the conversion impractical: The prototype chain and field sets on objects are both difficult to model in SMT. They can be modeled by enumerating all possible field names in the program and producing constraints over all pairs (τ, f) of type variables and fields, but this results in a $O(V * F)$ blowup in the number of constraints for systems with V type variables and F distinct field names. While only a polynomial increase, the large number of extra constraints necessary to model the system results in an unacceptable performance penalty. As a result, it was more practical to augment the existing SJSx type solver than to implement a new one on top of SMT, and we suspect this is also the case for other constraint-based type inference algorithms.

5. Adding Weights to Constraints

In general, there may be many valid correcting sets, reflecting that there are generally many ways to fix a typing error. Consider again the example from Section 2, reproduced here for convenience.

```
1  let f x = print_int x in
2  let g x = x + 1 in
3  let x = "hi" in
4    f x;
5    g x
```

Section 2 shows how MYCROFT generates a correcting set for this example containing a single constraint, generated from line 3, which equates the type of `x` with *string*. This correcting set suggests that assigning `"hi"` to `x` causes the type violation. Changing `x` to type *int* would resolve the type conflict, but this is not the only way to fix the program. Other correcting sets exist, such as the correcting set that contains constraints generated from lines 1 and 2, which corresponds to changing how the functions `f` and `g` use their arguments.

So far, we have implicitly assumed that every constraint is equally important and that smaller correcting sets are better, but this may not be the case in practice. Previous work has suggested attaching a weight to each generated constraint [20]. MYCROFT allows compiler writers to extend the type constraint generator to mark constraints as *hard* or *soft*, and to label soft constraints with weights indicating their relative importance in contributing to a type violation. This, in turn, allows the compiler writer to influence the selection of correcting sets.

Determining Weights: An Open Question. As we will see in Section 7, MYCROFT produces error reports of comparable quality to competing tools. It does this without using weights on soft constraints—all soft constraints are assigned equal weight—and with minimal use of hard constraints. MinErrLoc and SHErrLoc (also included in the evaluation) use different weighting schemes. Hence, it appears that the best means of weight selection remains an open question, and we hope to investigate a more systematic exploration of weight generation in the future. The remainder of this section describes how to extend MYCROFT with weighted constraints and how the OCaml and SJSx implementations divide constraints into hard and soft constraints.

5.1 Hard Constraints

Hard constraints are excluded from consideration when forming a correcting set. A compiler writer may mark some constraints as hard constraints when it is clear they should not be part of error explanation. MYCROFT uses hard constraints to steer error explanation away from "glue" constraints that connect conflicting uses and definitions. As an example, consider the following program.

```
1  let a = 1;;
2  let b = a;;
3  let c = b;;
4  print_string c;;
```

Our constraint generator produces constraints that equate `a` with `b` and `b` with `c`, but these are less interesting than the constraints relating `a` with `1` and `c` with `print_string`. Marking them as hard constraints causes MYCROFT to exclude them from candidate correcting sets.

MYCROFT also uses hard constraints to mark parts of the program that cannot be changed, such as the constraints generated from type signatures of library functions. In the example above, one of the generated constraints might be that "`print_string` has a type equal to `string` $\rightarrow$ `unit`." This is a hard constraint, since `print_string` is a library function outside of the programmer's control.

Requirements on Hard Constraints. Because hard constraints are, by definition, excluded from consideration by the error explanation engine, it is vital that they do not conflict amongst themselves. For the two uses of hard constraints

listed above, we found this requirement easy to satisfy. "Glue" constraints do not introduce any new typing information; rather, they propagate other constraints across type variables, and so glue constraints alone cannot conflict. Library functions have type annotations supplied in the context, and constraints that equate library function types with type variables are marked hard. But to generate conflicting constraints, library functions must actually appear in call sites, and call sites generate soft constraints.

5.2 Soft Constraints

Where hard constraints allow the compiler writer to concretely specify constraints that should not be part of a correcting set, weights on soft constraints assign relative importance between soft constraints. Neither the OCaml nor SJSx implementations make use of weights on soft constraints; rather, all constraints are assigned a constant weight of `1`.

5.3 Extending MYCROFT with Weighted Constraints

Given a program that produces a set of hard constraints $\mathcal{A}_H$, soft constraints $\mathcal{A}_S$, and weights w, the algorithm in Figure 6 solves a combinatorial optimization problem similar to Equation 8 by searching for a fix $\mathcal{F}$ such that

$$\underset{\mathcal{F} \subseteq \mathcal{A}_S \wedge sat(\mathcal{A}_H \cup \mathcal{A}_S - \mathcal{F})}{\arg\min} \sum_{f \in \mathcal{F}} w(f) \tag{9}$$

where sat stands for the type solver and determines whether a set of constraints is satisfiable, and w is a weighting function for each constraint. In short, it seeks a minimal-weight correcting set $\mathcal{F}$ from among the soft constraints such that $\mathcal{A}_H \cup \mathcal{A}_S - \mathcal{F}$ is satisfiable.

Figure 6 shows the core MYCROFT algorithm first presented in Figure 1 extended to account for weighted constraints. The constraint generator (`TYPECGEN.Generate`) on line 2 now produces hard constraints $\mathcal{A}_H$, soft constraints $\mathcal{A}_S$, and a weighting function w over the soft constraints. Line 3 enforces the requirement that hard constraints be satisfiable, and then `FIND_FIX` is invoked. `FIND_FIX` is largely unchanged. Weights influence the selection of the correcting set $\mathcal{S}'$ on line 12 produced by `FindCandSet`.

Section 3.3 presents two mechanisms for selecting a candidate correcting set: MaxSAT and set cover. Conveniently, both mechanisms extend naturally to handle weighted constraints, in the form of weighted MaxSAT and weighted set cover. Using either implementation, `FindCandSet` will return a minimally-weighted candidate correcting set.

5.4 Revisiting the Metatheory

Minimality (Theorem 2) also holds for MYCROFT extended with weighted constraints: In this case, "minimal" is taken to mean "least aggregate weight" instead of "smallest size." However, the termination argument (Theorem 1) must be extended to account for hard constraints.

The termination argument relies on two properties of MYCROFT: The candidate correcting sets produced by

```
1  MYCROFT(p) =
2    let w, A_H, A_S = TYPECGEN.Generate(p)
3    if TYPESOLVER.Solve(A_H) = unsat then fail
4    else return FIND_FIX(w, A_H, A_S, [], ∅)
5
6  FIND_FIX(w, A_H, A_S, L, F) =
7    let C = (A_H ∪ A_S) - F
8    if TYPESOLVER.Solve(C) = sat
9    then return F
10   else let U = TYPESOLVER.UnsatCore()
11        let L' = U::L
12        let F' = FindCandSet(w, A_H, A_S, L')
13        return FIND_FIX(w, A_H, A_S, L', F')
```

Figure 6. The MYCROFT algorithm with weighted constraints.

`FindCandSet` are unique, and constructing a candidate set containing every constraint will succeed. The latter is no longer necessarily true when weighted constraints are introduced. Hard constraints cannot be removed, and so a candidate correcting set that contains every soft constraint will still fail if the hard constraints conflict amongst themselves. We solve this by requiring that the hard constraints be satisfiable. As Section 5.1 discusses, this is not an onerous burden.

6. Implementation

The core MYCROFT algorithm is implemented as a library in Java with 662 lines of code. It is shared by the OCaml and SJSx implementations, and it includes two versions of the function `FindCandSet()` of Figure 1, which selects candidate correcting sets. The first is a greedy approximation of the weighted set cover problem, and the second employs MaxSAT by calling out to Sat4j [16].

The changes necessary to instrument each solver were relatively small compared to the sizes of the solvers, reflecting the relative ease of adopting MYCROFT as an error explanation engine. We augmented the OCaml compiler with 666 lines of code to produce type constraints. The type solving algorithm shown in Figure 4c was implemented in 985 lines of Java. We estimate that less than 100 lines of code in the solver are related to unsat core tracking, but because unsat core tracking is a cross-cutting concern, it is difficult to measure this precisely. We added 366 and altered 531 lines of code to extend the SJSx constraint generator to annotate constraints with weights and extend the SJSx solver to produce unsat cores. The magnitude of the required changes is small compared to the size of the SJSx type inference engine, which totals more than more than 9500 lines.

7. Evaluation

We have evaluated MYCROFT along several dimensions:

- **Error Localization Quality** (Section 7.1): Compared to other state of the art tools, how often does MYCROFT find the correct location of a type error? We found that MYCROFT localizes errors as well as previous approaches.
- **Performance** (Section 7.2): Compared to other state of the art tools, how quickly does MYCROFT produce a complete error report? While previous approaches require several minutes to analyze files just 400 lines long and time out after 30 minutes on a 1000 line program, we found that MYCROFT can produce reports for files over 1000 lines long in less than 5 seconds.
- **Greed vs. MaxSAT** (Section 7.3): While MYCROFT runs much faster in greedy approximation mode, is it worth sacrificing minimality in correcting sets? We found that, in practice, the greedy error report had comparable quality to the MaxSAT report, and thus we believe the trade-off is worthwhile.
- **Qualitative Report Quality** (Section 7.4): Is MYCROFT useful in practice? We report on our experience porting an open-source JavaScript implementation of *annex*, an Othello-like game, to fit within the SJSx typing discipline. As *annex* is not written in a style immediately typeable with SJSx, this effort offered extensive interaction with type inference error messages. We had positive experiences with MYCROFT, and MYCROFT is currently the default error explanation algorithm for SJSx.

7.1 Error Localization Quality

In this experiment, we compare the accuracy of type error feedback obtained by MYCROFT against other state of the art approaches. Our subject programs are student programming assignments, generously shared with us by Lerner *et al*. [17]. The assignments are all short OCaml programs collected while the students were working, and most exhibit some defect such as a syntax error or one or more type errors.

SHErrLoc and MinErrLoc. We compare MYCROFT against two state of the art tools: SHErrLoc and MinErrLoc.[4] Section 8 discusses these and other approaches against a broader background of related work, but a brief overview of the tools themselves will help in discussing the results of our evaluation.

At a high level, the three tools are similar: Each converts the program into a set of type constraints and attempts to find a constraint subset that optimizes some criterion. However, each tool optimizes for a different criterion. MYCROFT optimizes Equation 9 using a constant weight of 1 for each soft constraint, but marks some constraints as hard. Specifically, MYCROFT's hard constraints encode (1) "glue" constraints that relate let-bound variables to their definitions and (2) typing constraints from other modules. MinErrLoc optimizes Equation 9 using $w(f) =$ size of AST node associated with f. MinErrLoc also uses hard constraints, but in a slightly different manner: MinErrLoc's hard constraints encode (1) global properties of the OCaml type system required to formulate the problem for its underlying SMT solver, (2) typing constraints from other modules, and (3) user-defined type annotations. SHErrLoc optimizes for a similar formula that factors in the number of correct uses of each type variable (see Section 8.2), and does not use hard constraints.

SHErrLoc also employs *preemptive cutting* [20]: It stops generating constraints when the first type error is discovered. Constraints that appear later in the file—which may be relevant to the error—are ignored. As a result, SHErrLoc is unable to discover all type errors in some files. MinErrLoc supports preemptive cutting as an optional feature, but it is disabled for our experiments. Perhaps surprisingly, we found these differences to have little impact on explanation quality.

	Perfect	Precision	Recall	Fail
ocamlc	27	84%	40%	0
SHErrLoc	23	72%	52%	12
MinErrLoc	27	70%	62%	6
MYCROFT (MaxSAT)	29	75%	77%	5
MYCROFT (Greedy)	28	63%	67%	4

Table 1. Comparison of error reporting quality on 50 tests. Tools can fail on some files due to timeout or unsupported language features ("Fail").

	Perfect	Precision	Recall
ocamlc	19	84%	50%
SHErrLoc	20	72%	51%
MinErrLoc	20	70%	56%
MYCROFT (MaxSAT)	21	76%	72%
MYCROFT (Greedy)	21	71%	69%

Table 2. Comparison of error reporting quality on 32 tests, excluding tests on which any tool failed.

Methodology. Lerner *et al*. have already categorized the programs in the dataset according to the nature of the defects. These categorizations include "invalid syntax," "type mismatch," "unbound variable," and many others.

To obtain ground-truth data for correct error locations, we randomly selected 50 files from among those categorized with type-related defects. For each one we identified a set of locations that—in our best judgment—corresponded to places at which there is a fixable type error.[5] We refer to our hand-identified locations as the *oracle data* for the file.

[4] We are using the 2014 versions of SHErrLoc and MinErrLoc. While newer results were published for both tools in 2015, the newer versions are not publicly available.

[5] Previous work that used this data set also did this exercise, but unfortunately this information was not available.

Notably, the authors of SHErrLoc used the location of each student's next changes as oracle data [26]. While we referred to this data to help decide where to mark oracle error locations, we found it to be too noisy to serve as reliable oracle data on its own. The students often made many changes between compiler runs, not all of which were related to the type errors.

Having collected oracle data, we ran each tool to obtain its set of reported locations on each file, which we then compared to the oracle data. For some files, one or more of the tools could not be run successfully (either due to features that the tool does not support or because it exceeded our imposed timeout of 180 seconds). Out of 50 files, 32 succeeded with every tool. We report aggregate quality information for both the complete set of 50 and the partial set of 32, though the results are not substantially different.

Results. Table 1 shows the aggregate precision and recall for each tool across the 50 files for which we derived oracle data. For any particular tool and file, precision is the percentage of program locations reported by the tool that are true error locations according to the oracle data:

$$precision = \frac{|tool\_locs \cap oracle\_locs|}{|tool\_locs|}$$

Recall is the percentage of oracle data locations that were reported by the tool:

$$recall = \frac{|tool\_locs \cap oracle\_locs|}{|oracle\_locs|}$$

The "Perfect" column reports the number of files for which each tool scored 100% on both precision and recall. The "Fail" column reports how many files the tool rejected due to unsupported language features or timeout.

The numbers in Table 1 are not directly comparable since different tools fail on different sets of files. To make a more direct comparison possible, we also report on the 32 cases for which all the tools completed successfully (Table 2).

Our complete results table—from which Table 1 and Table 2 are derived—is shown in the appendix.

Details and Discussion. MYCROFT is competitive with SHErrLoc and MinErrLoc, the other two state of the art approaches. The OCaml compiler performs very differently from the other tools since it is limited to only producing one type error at a time. While it shows high precision across our benchmark files, this number alone can be misleading: ocamlc rarely produces a perfect error report.

In general, the tools we evaluated do well or poorly together; there are very few programs in the Seminal dataset on which any one tool greatly outperforms the others. The few differences tend to arise from differences in constraint generation or when the underlying optimization algorithms choose arbitrarily between more than one equal-cost set, rather than any fundamental attribute of the approach.

```
1  type elt = IntEntry of int | HeapEntry of heap
2
3  let rec lookup h str =
4    match h with
5    | [] → 0
6    | hd::tl → match hd with
7        | IntEntry (str1,i) →
8          if str1=str then i
9          else lookup h str
10       | HeapEntry(str2,j) →
11         if str2=str then j
12         else lookup tl str
13
14 let update h str i =
15   match i with
16   | int  → IntEntry(str,i)::h
17   | heap → HeapEntry(str,i)::h
```

(a) A benchmark for which MYCROFT performs perfectly.

```
1  type move = (* ... *) | For of int * move list
2  let makePoly sides len =
3    For(sides, [(* ... *)])
4
5  let interpLarge (ml : move list) :
6    (float*float) list = (* ... *)
7
8  let example_logo_prog = makePoly(4, 10)
9  let ansL = interpLarge example_logo_prog
```

(b) A benchmark for which MYCROFT performs poorly.

Figure 7. Example programs from the Seminal dataset with correct error locations highlighted.

As an example, consider the program in Figure 7a that MYCROFT analyzes correctly, while the OCaml compiler, SHErrLoc, and MinErrLoc miss one or more error sources. The yellow highlighted lines are errors: The `hd` variable comes from a list of `string * elt` tuples, and so the first branch, for instance, should instead be `(str1, IntEntry i)`. Returning `j` in the second branch is also problematic, since `j` is a `heap` and not an `int`. The `update` function, defined later in the file, also contains two similar errors, where the student mistakenly constructed an `IntEntry` and a `HeapEntry` out of tuples.

MYCROFT reports errors at each highlighted location. SHErrLoc stops generating constraints as soon as an error is detected, and so is unable to identify the last two locations. MinErrLoc reports two extraneous locations: the `hd` in `match hd` and a use of `update` lower in the file (not shown above).

Sometimes MYCROFT does poorly while SHErrLoc and MinErrLoc do well, as in the case of Figure 7b. In this example, the student has defined a function `makePoly` that returns a `move`, but the function is passed to `interpLarge`, which expects a `move list`. Our oracle data for this file was based on the student's fix: the student changed the use of `makePoly` on line 8 to `[makePoly 4 10.0]`.

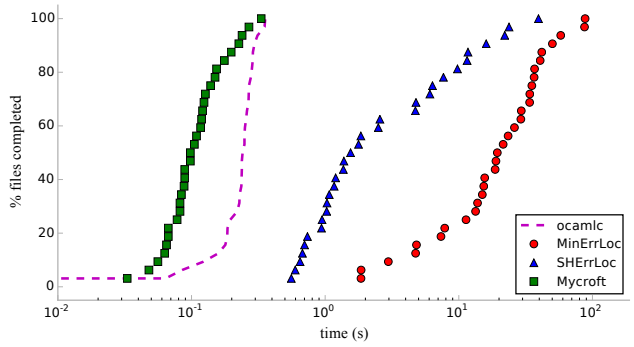

Figure 8. Run-time to analyze the Seminal files.

LoC	Line	Total time (s) SHErrLoc	MinErrLoc	MYCROFT
91	87	0.5	1.7	0.1
121	100	0.6	2.3	< 0.1
181	137	1.2	2.7	< 0.1
242	167	1.8	3.5	0.1
336	208	42.0	8.0	0.1
466	269	-	25.8	0.4
687	399	-	-	0.3
805	437	-	-	0.3
970	522	-	-	0.4
1234	663	-	-	0.6
1327	701	-	-	0.6

Table 3. Scalability data on prefixes of the min-rt program with a 180s timeout.

SHErrLoc and MinErrLoc correctly identify the second-to-last line, while MYCROFT chooses to report the definition of `makePoly` instead. Since `makePoly` is only used once, there is no factor in MYCROFT's cost function (Equation 9) to prefer reporting the *definition* of `makePoly` or its *use* as the source of the error.

7.2 Performance

Speed is a key benefit of the architecture used in MYCROFT; in experiments we performed over the Seminal dataset and on a larger OCaml program, MYCROFT consistently outperforms SHErrLoc and MinErrLoc.

Methodology. Our performance evaluation consists of two parts. First, while measuring the quality data reported in Section 7.1, we also recorded the time each tool takes to analyze each file, which gives a sense of the distribution of run-time performance on small files.

Excluding comments and blank lines, none of the Seminal files have more than 100 lines of code. To investigate scalability on larger programs, we obtained a 1,300-line ray-tracing program called min-rt [19]. Min-rt is an appealing evaluation benchmark because it uses only simple features of the OCaml language and so fits within the supported subset of every tool we evaluate. We construct a number of programs of increasing length by taking prefixes of the main min-rt file and introducing a simple arithmetic type error in each prefix program at approximately the middle line of the file. We then measure the wall-clock time each tool takes to analyze each prefix. Experiments were run on a shared compute server with 96 cores and 512 Gb of memory. The Java heap limit was set to 64 Gb to allow running the evaluation in parallel; in practice, MYCROFT does not require more than 11 Mb of memory on even the largest files in our evaluation.

Performance on the Seminal Dataset. Figure 8 shows performance information on the Seminal dataset, plotting total elapsed time (x-axis, log scale) against the percentage of the selected Seminal tests completed—farther left is better. For a closer comparison with the other tools, we evaluated MYCROFT's MaxSAT strategy. Even on these small files, MYCROFT exhibits faster and more consistent performance than SHErrLoc or MinErrLoc. The increased performance is the result of MYCROFT's specialized constraint system and solver.

Performance on Min-rt. Table 3 shows the total lines (including blank lines and comments) of each min-rt prefix, as well as the line mutated to induce an error, and the wall-clock time each tool takes to analyze the file. A dash indicates entries that timed out, either during constraint generation or analysis. In this experiment we use MYCROFT's MaxSAT strategy. In all cases, every tool that completed on each file found the correct error location.

The performance numbers show that total explanation time grows very quickly for SHErrLoc and MinErrLoc. MYCROFT, on the other hand, manages to construct a report for each prefix in less than one second. Even with a 30 minute timeout, neither SHErrLoc nor MinErrLoc complete on the full 1327 line version of min-rt.

Impact of Constraint Generation on Performance. Although the performance results—particularly the min-rt results—show MYCROFT's potential, they also highlight a limitation in the SHErrLoc and MinErrLoc implementations. Both tools are prone to generating a large number of constraints in the presence of polymorphism: Because neither constraint language includes generalized polymorphic types directly, each tool duplicates the constraints of every polymorphic function at each application site. This problem is a symptom of the mismatch between these tools' constraint languages the type system they are modeling. Since MYCROFT is agnostic to the underlying constraint system and solver, we avoid this problem by supplying it with a custom constraint system (Section 4.2) that represents polymorphic types directly rather than eagerly duplicating constraints.

Although the scalability of SHErrLoc and MinErrLoc can be improved with more exotic techniques [21], our results show that MYCROFT is quite competitive, in part because

Errs	MaxSAT (s)	Greedy (s)
0	0.9	1.2
2	1.1	1.3
4	1.6	1.5
6	4.0	2.6
8	109.5	2.6
10	-	3.9

Table 4. Performance of MaxSAT vs. greedy strategies as the number of independent type errors ("Errs") grows.

MYCROFT can take advantage of performance improvements developed directly for type inference.

7.3 Was Greedy Computation Important?

Despite the appeal of a minimal correcting set, we found that a greedy strategy for selecting candidate correcting sets performs much faster than MaxSAT without sacrificing much quality. Tables 1 and 2 include results of error message quality using MYCROFT with both implementations. The greedy approach matched the oracle nearly as well as the MaxSAT approach and just as well as SHErrLoc and MinErrLoc.

Correcting set selection scales with the number of independent errors in the file—*i.e.* the number of unsat cores the algorithms must consider. To investigate performance, we introduced additional mutations to min-rt, each introducing an independent type error. Table 4 compares how long the MaxSAT and greedy strategies take to produce an error report for each mutation. The MaxSAT implementation times out when there are more than eight independent errors, but the greedy implementation can produce an error report in less than 5 seconds in all cases.

Using a greedy solver by default was especially useful in our SJSx case study. In that domain, a common use-case is porting an ill-typed JavaScript program to well-typed SJSx. Thus, starting with dozens of type errors was a normal state of affairs.

7.4 MYCROFT and SJSx

We now come full circle to our starting point: We needed a good error explanation for SJSx programs, especially when refactoring untyped JavaScript code to be compliant with the SJSx type system (which is a strict subset of JavaScript).

We have ported many programs to type check with SJSx, and in a few cases we preserved the intermediate versions as the developer iteratively resolved type errors. Unfortunately, this porting was carried out without the benefit of MYCROFT; the type inference engine would abort at the very first conflict. This was not a good experience for the programmer.

After developing MYCROFT, we revisited the intermediate snapshots to see how MYCROFT would have performed on those versions. We present the results of our investigation on *annex*, which is an Othello-like game. Beginning with the unmodified JavaScript (without GUI support, so as to run on node.js), our goal was to bring *annex* to a state that would type check with SJSx.

commit	time(s) greedy	errors greedy	time(s) maxsat	errors maxsat
1	36.2	14	-	-
2	19.2	14	-	-
3	19.4	14	-	-
4	18.5	12	-	-
5	15.1	10	190.0	10
6	13.3	7	18.0	7
7	12.7	6	16.0	6
8	14.7	6	22.3	7
9	10.0	4	9.6	4
10	8.0	1	7.0	1

Table 5. Running MYCROFT on snapshots of *annex* under porting.

In one of the earliest *annex* snapshots, the old engine (before MYCROFT) would stop with the following unhelpful message ("meet" is a type inference operation in SJSx, and the "X" variables are internal type variables):

```
cannot meet object type {  | instance: X833 } with
   constructor type ctor<0>[X9]() -> X8
```

MYCROFT produced the following output (shown in part) on the same snapshot:

```
Error on line 530: read of missing property 'step'
    Note: type is { evaluate: ___, ... |  }

Error on line 544: wrong key type used on Array<___>
    Note: type is string, expected integer

Error on line 544: wrong key type used on Array<string>
    Note: type is string, expected integer
```

As in the min-rt experiment, we found the greedy computation of correcting sets to be important for practicality. Table 5 shows the performance of MYCROFT on 10 snapshots of *annex*. We find that in the starting snapshots—when the correcting set is large—the greedy algorithm is crucial for MYCROFT to run to completion. When the correcting set is small, greedy and MaxSAT perform comparably. Table 4 also confirms these results. Our experience with other programs was similar. At this time, MYCROFT (with the greedy explanation strategy) is the default error explanation engine in the SJSx compiler.

7.5 Limitations and Future Work

MYCROFT is designed to work with type inference algorithms that naturally decompose into type constraint generation and solving. Although the SJSx type inference algorithm mostly follows this design pattern, it adds a third stage, after solving, where it checks that certain implicit constraints hold. For example, the third stage checks whether, for each field access,

the object so accessed possesses the field. Failure at this stage still results in a typing violation.

As a result, we modified the SJSx constraint system to turn some of these implicit constraints into explicit constraints. With additional visibility, MYCROFT can do a better job of diagnosing type errors. Other constraint-based error explanation tools have made the same observation; they carefully craft their constraint systems to capture the behavior they hope to report on [11, 20].

As discussed in Section 7.1, while there may be many minimum-size correcting sets for a given program, MYCROFT has no heuristics for selecting between them. Picking the right weights to best differentiate between correcting sets is still an open problem (Section 5).

Finally, we found that the comprehensiveness of the correcting set is not necessarily important to a programmer; if the set is large, he or she would typically scan a few findings, fix whatever looks straightforward, and run the tool again. This suggests a line of future work exploring a ranking mechanism to highlight the most important or easiest-to-fix program points in the correcting set.

8. Related Work

MYCROFT builds on research for type inference, type error diagnosis, and constraint solving.

8.1 Type Inference

Type inference is the task of determining the types of program expressions in the absence of type annotations. It has been studied extensively for languages with Hindley-Milner (HM) style type systems [7, 23]. In particular, our conversion from OCaml programs to constraints (Figure 4b) is directly inspired by previous work on understanding HM type inference in terms of constraints [25, 18, 11, 1]. Our use of polymorphic terms in the constraint language is atypical but was inspired by previous work on instantiation constraints [22, 12, 8].

8.2 Type Error Diagnosis

Many modern languages have support for type inference, but providing good error messages for ill-typed programs is a famously hard problem with various proposed solutions.

Slicing. Type error slicing is the task of identifying all program locations relevant to a particular type error [11]. When presenting a type error slice to the programmer, a key problem is how to minimize the size of the slice to avoid presenting too much irrelevant information. Unsat cores are analogous to type error slices, but since the unsat cores are not shown to the programmer, MYCROFT does not require core minimization.

Source-Level Fixes. The Seminal system [17] proposes concrete source-level changes to fix a type error. Seminal uses the type checker as a black box and intelligently searches for mutations to the program source that allow it to typecheck. A set of heuristics prevents degenerate solutions such as "delete the whole program." When Seminal's proposed fix is correct, the programmer can simply copy the fix into her own program. However, there are many potential source changes that satisfy the type checker, and suggesting the wrong one is useless to the programmer—finding the right set of rewriting heuristics is hard and must be repeated for each new language.

Type-Level Fixes. More recent systems suggest type-level fixes: They point to specific program locations and explain in terms of types why something is wrong with that location. To make this concrete, for the ill-typed expression `not 1`, Seminal might suggest "replace `1` with `false`" while a type-level fixer might suggest "`1` has type `int`, but a `bool` is expected here." Not committing to a specific program change can be an advantage—for instance, if the programmer intended `true` instead of `1`—but both approaches solve the *localization* problem by pointing to a specific program expression.

There are three major recent lines of work investigating type-level fixes: counterfactual typing, SHErrLoc, and MinErrLoc. Chen and Erwig [4] present the idea of counterfactual typing. Their work attempts to discover potential type-level fixes based on "variational" typing, which can track type judgements over a space of alternatives and explore variations that would side step type failures. They offer the user a ranked list of suggestions, starting with type fixes on single expressions, to fixes on larger expressions. Unlike MYCROFT, counterfactual typing is highly language dependent and has only been studied for HM type systems.

SHErrLoc is a constraint solver which is applicable to several static analysis diagnosis tasks [26, 27]. SHErrLoc finds a fix $\mathcal{F}$ for type constraints $\mathcal{C}$ according to

$$\underset{\mathcal{F} \subseteq \mathcal{C} \wedge sat(\mathcal{C}-\mathcal{F})}{\arg\min} \quad C_1 \left|\mathcal{F}\right| + C_2 \kappa(\mathcal{F}, \mathcal{C})$$

where C_1, C_2 are positive constants. The function $\kappa(\mathcal{F}, \mathcal{C})$ counts the number of times that terms in $\mathcal{F}$ appear on "satisfiable paths" in $\mathcal{C}$, and captures the intuition that terms used correctly in many places are unlikely to be sources of error. Unfortunately, the notion of satisfiable paths is difficult to define for complex type systems like SJSx. Even for OCaml, computing $\kappa(\mathcal{F}, \mathcal{C})$ can be expensive.

MinErrLoc explains OCaml type errors by converting type constraints into a form that can be solved by an off-the-shelf SMT solver [20, 21]. MinErrLoc implements a MaxSMT solver, allowing it to find a correcting set according to Equation 9. MYCROFT generalizes these ideas to arbitrary constraint systems and solvers and shows that a greedy approximation yields substantial performance improvement without significant loss in error reporting quality.

Dynamic Witnesses. Since static type checkers exist to prevent run time errors, an input that produces a run time error is a sensible explanation for the type error. Seidel *et al*. [24] propose a diagnosis system that finds concrete inputs (witnesses) that lead to bad executions. This approach is not total; it is only able to find witnesses in 88% of the

student programs it was evaluated on. However, it nicely complements error localization approaches like MYCROFT, as the two can be used side-by-side.

8.3 Constraint Systems

The task of finding a minimal-weight correcting set is the complement of the *maximum weighted satisfiability problem*: Given a set of constraints, find a maximum-weight subset that is satisfiable. Many modern MaxSAT solvers use an unsat core driven approach similar to the one we have described [9].

Unsat cores are also useful in the DPLL(T) framework for solving SMT formulas [10, 2]. As a result, unsatisfiable core extraction has been studied for a number of different constraint systems, including equality with uninterpreted functions and linear arithmetic. The QuickXplain algorithm [13] can even produce unsat cores for black-box constraint solvers, and was used successfully by MinErrLoc [20].

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM.

[2] R. Bruttomesso, E. Pek, and N. Sharygina. A flexible schema for generating explanations in lazy theory propagation. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 41–48, July 2010.

[3] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-il Choi. Type inference for static compilation of JavaScript. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '16, New York, NY, USA, 2016. ACM.

[4] Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM.

[5] Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. SJS: A type system for JavaScript with fixed object layout. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2015.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[8] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.

[9] Zhaohui Fu and Sharad Malik. On solving the Partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer Berlin Heidelberg, 2006.

[10] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, chapter DPLL(T): Fast Decision Procedures, pages 175–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[11] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Conference on Programming*, ESOP'03, pages 284–301, Berlin, Heidelberg, 2003. Springer-Verlag.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, April 1993.

[13] Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.

[14] Richard M. Karp. *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, chapter Reducibility among Combinatorial Problems, pages 85–103. Springer US, Boston, MA, 1972.

[15] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, March 1994.

[16] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[17] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM.

[18] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, January 1999.

[19] Yutaka Oiwa and Eijiro Sumii. min-rt. `https://github.com/esumii/min-caml/blob/master/min-rt`.

[20] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM.

[21] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Practical SMT-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 412–423, New York, NY, USA, 2015. ACM.

[22] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin Pierce, editor, *Advanced Types and Programming Languages*. MIT Press, 2005.

[23] Didier Rémy. Extending ML type system with a sorted equational theory. Technical report, Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[24] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors. *CoRR*, abs/1606.07557, 2016.

[25] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical report, 1999.

[26] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 569–581, New York, NY, USA, 2014. ACM.

[27] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 12–21, New York, NY, USA, 2015. ACM.

A. Reduction to Set Cover

Definition 2 (The hitting set problem (a.k.a. set overlap))**.** *Suppose we have a finite set $\mathcal{X}$ and a family $\mathcal{F}$ of non-empty subsets of $\mathcal{X}$ such that every element of $\mathcal{X}$ belongs to at least one subset in $\mathcal{F}$:*

$$\mathcal{X} = \bigcup_{\mathcal{S} \in \mathcal{F}} \mathcal{S}$$

A set $\mathcal{H} \subseteq \mathcal{X}$ is a hitting set *if it has a non-empty intersection with each $\mathcal{S} \in \mathcal{F}$:*

$$\forall \mathcal{S} \in \mathcal{F}.\, \mathcal{H} \cap \mathcal{S} \neq \emptyset$$

The problem is to find the smallest such hitting set.

The set overlap problem is a variant of the *set covering* problem [14], where the task is to find a minimal-sized subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of $\mathcal{X}$.

Theorem 3. *Set overlap is NP-hard.*

Proof. The proof proceeds by reducing set cover to set overlap. Suppose we have a finite set $\mathcal{X}_c$ and a family $\mathcal{F}_c$ of subsets of $\mathcal{X}_c$ as an instance of the set cover problem. Let $\mathcal{X}$ be a set of unique labels for the sets in $\mathcal{F}_c$, and let a set $\mathcal{S}$ be in $\mathcal{F}$ for each $x \in \mathcal{X}_c$ such that $\mathcal{S}$ contains the labels for each set in $\mathcal{F}_c$ that contains x. Note that $\mathcal{X}$ can be constructed in $\mathcal{O}(\mid \mathcal{F}_c \mid)$ and $\mathcal{F}$ in $\mathcal{O}(\mid \mathcal{F}_c \mid \cdot \mid \mathcal{X}_c \mid)$.

$$\mathcal{X} = \{i \mid \forall i.\, \mathcal{S}_i \in \mathcal{F}_c\}$$
$$\mathcal{F} = \{\{i \mid \forall i.\, \mathcal{S}_i \in \mathcal{F}_c \wedge x \in \mathcal{S}_i\} \mid \forall x.\, x \in \mathcal{X}_c\}$$

The set overlap problem produces a solution $\mathcal{C}$ such that

$$\forall \mathcal{S} \in \mathcal{F}.\, \mathcal{C} \cap \mathcal{S} \neq \emptyset.$$

That is, $\mathcal{C}$ is the smallest set of labels such that at least one label appears in each $\mathcal{S} \in \mathcal{F}$. From this we construct a minimal cover

$$\mathcal{C}_c = \{\mathcal{S}_i \mid \mathcal{S}_i \in \mathcal{F}_c \wedge i \in \mathcal{C}\}.$$

To complete the proof, we show that $\mathcal{C}$ is a minimal overlap if and only if $\mathcal{C}_c$ is a minimal cover.

If $\mathcal{C}$ is a minimal overlap, then $\mathcal{C}_c$ is a minimal cover. Note that $\mathcal{X}$ holds the sets from $\mathcal{F}_c$ that can make up a cover, and each set $\mathcal{S} \in \mathcal{F}$ is the set of sets in $\mathcal{F}_c$ that cover a particular element $x \in \mathcal{X}_c$. The set overlap problem picks at set $\mathcal{C}$ of elements from $\mathcal{X}$ such that $\mathcal{C}$ has a non-empty intersection with every set in $\mathcal{F}$. Critically, for a set $\mathcal{S} \in \mathcal{X}$ to overlap with a set $\mathcal{S}_i \in \mathcal{F}$ implies that $x_i \in \mathcal{S}$. As $\mathcal{F}$ contains a set $\mathcal{S}_i$ for each $x_i \in \mathcal{X}_c$, we have that the sets in $\mathcal{C}$ are exactly the sets of $\mathcal{F}_c$ that cover the elements of $\mathcal{X}_c$.

But suppose a smaller cover $\mathcal{C}'_c$ exists. We can construct

$$\mathcal{C}' = \{i \mid \mathcal{S}_i \in \mathcal{C}'_c\}.$$

As $\mathcal{C}'_c$ is smaller than $\mathcal{C}_c$, so too is $\mathcal{C}'$ smaller than $\mathcal{C}$. But we assume that $\mathcal{C}$ is a minimal overlap, which implies that there exists $\mathcal{S}_i \in \mathcal{F}$ where $\mathcal{C}' \cap \mathcal{S}_i = \emptyset$. By construction, this indicates that no set $\mathcal{S} \in \mathcal{C}'_c$ contains x_i, which is a contradiction.

If $\mathcal{C}_c$ is a minimal cover, then $\mathcal{C}$ is a minimal overlap. If $\mathcal{C}_c$ covers $\mathcal{X}_c$, then for all $x \in \mathcal{X}_c$, a set $\mathcal{S}$ exists such that $x \in \mathcal{S} \in \mathcal{C}_c$. By construction, each $x \in \mathcal{X}_c$ has a corresponding set $\mathcal{S}_x \in \mathcal{F}$ such that $\mathcal{S} \in \mathcal{S}_x$. Therefore $\mathcal{C}$, which contains a label i for $\mathcal{S}$, will overlap with $\mathcal{S}_x$. This holds for all $x \in \mathcal{X}_c$, hence $\mathcal{C}$ overlaps with all $\mathcal{S}_x$.

But suppose a smaller overlap $\mathcal{C}'$ exists. Then a corresponding $\mathcal{C}'_c$ exists smaller than $\mathcal{C}_c$; but $\mathcal{C}_c$ is assumed to be a minimal covering, and so there exists an element $x \in \mathcal{X}_c$ that $\mathcal{C}'_c$ does not cover. By construction, there exists a set $\mathcal{S}_x \in \mathcal{F}$ containing exactly the labels of sets in $\mathcal{F}_c$ that cover x, but none are present in $\mathcal{C}'_c$ and thus not in $\mathcal{C}'$, which implies that $\mathcal{C}' \cap \mathcal{S}_x = \emptyset$, a contradiction.

□

B. Quality Data

Table 6 shows the complete report of the summarized data presented in Table 1 and Table 2.

file hash	ocamlc		SHErrLoc		MinErrLoc		MYCROFT (MaxSAT)		MYCROFT (Greedy)	
	precision	recall	precision	recall	precision	recall	precision	recall	precision	recall
a64278e6	1 / 1	1 / 3	1 / 1	1 / 3	1 / 1	1 / 3	2 / 3	2 / 3	2 / 3	2 / 3
85a6929f	1 / 1	1 / 2	timeout		2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2
9794c8f8	1 / 1	1 / 1	0 / 0	0 / 0	0 / 1	0 / 1	error		error	
0c9a7476	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
47b33057	0 / 1	0 / 1	error		0 / 1	0 / 1	0 / 1	0 / 1	1 / 1	1 / 1
72b8814e	1 / 1	1 / 1	error		1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
6b0462a3	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
4a9f1fa2	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
d373092e	1 / 1	1 / 1	0 / 0	0 / 0	0 / 1	0 / 1	1 / 1	1 / 1	1 / 1	1 / 1
fde52031	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
b5136659	0 / 1	0 / 1	0 / 1	0 / 1	1 / 1	1 / 1	0 / 1	0 / 1	1 / 1	1 / 1
996270a2	1 / 1	1 / 1	error		1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
70bb62db	1 / 1	1 / 1	timeout		1 / 1	1 / 1	0 / 1	0 / 1	0 / 1	0 / 1
e583f71c	1 / 1	1 / 1	1 / 1	1 / 1	timeout		1 / 1	1 / 1	1 / 1	1 / 1
cb82b8e1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	0 / 1	0 / 1
94e25207	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
ae971991	1 / 1	1 / 1	1 / 1	1 / 1	0 / 1	0 / 1	1 / 2	1 / 1	0 / 2	0 / 1
364e8a47	1 / 1	1 / 1	timeout		1 / 1	1 / 1	1 / 1	1 / 1	0 / 1	0 / 1
f6a4b5a7	1 / 1	1 / 1	1 / 1	1 / 1	timeout		1 / 1	1 / 1	1 / 1	1 / 1
b168bdf6	1 / 1	1 / 3	timeout		timeout		3 / 3	3 / 3	3 / 3	3 / 3
c6e8c52f	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1
8a106f89	1 / 1	1 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2
06de74a4	1 / 1	1 / 3	1 / 1	1 / 3	3 / 3	3 / 3	3 / 4	3 / 3	3 / 4	3 / 3
4ea7768a	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
f4603d9b	1 / 1	1 / 16	timeout		timeout		timeout		10 / 19	10 / 16
e006ab3e	0 / 1	0 / 7	timeout		timeout		6 / 8	6 / 7	4 / 8	4 / 7
4385ca75	1 / 1	1 / 1	0 / 1	0 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
7148e940	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 2	1 / 1
644d210e	0 / 1	0 / 6	0 / 3	0 / 6	0 / 3	0 / 6	0 / 3	0 / 6	0 / 3	0 / 6
89b393b6	1 / 1	1 / 1	0 / 1	0 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
14e1c11a	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
2bd355fe	1 / 1	1 / 4	2 / 2	2 / 4	2 / 5	2 / 4	error		error	
4deca972	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
e39cffd4	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
62b3d2f6	1 / 1	1 / 2	1 / 1	1 / 2	0 / 1	0 / 2	0 / 1	0 / 2	0 / 1	0 / 2
6cf1158b	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
5d42f721	1 / 1	1 / 6	1 / 1	1 / 6	1 / 1	1 / 6	5 / 5	5 / 6	4 / 5	4 / 6
9fc4f621	0 / 1	0 / 2	0 / 2	0 / 2	0 / 1	0 / 2	0 / 1	0 / 2	0 / 1	0 / 2
efd02f8c	1 / 1	1 / 2	0 / 0	0 / 0	0 / 1	0 / 2	2 / 3	2 / 2	1 / 3	1 / 2
c23ed191	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
e429466f	1 / 1	1 / 4	2 / 2	2 / 4	4 / 6	4 / 4	4 / 4	4 / 4	4 / 4	4 / 4
c75f0b50	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	1 / 1	1 / 1
fbae0542	1 / 1	1 / 4	timeout		4 / 5	4 / 4	3 / 5	3 / 4	3 / 5	3 / 4
8708affd	1 / 1	1 / 3	timeout		3 / 3	3 / 3	3 / 6	3 / 3	0 / 6	0 / 3
7409001b	1 / 1	1 / 1	1 / 1	1 / 1	0 / 1	0 / 1	1 / 1	1 / 1	1 / 1	1 / 1
5ba718cb	1 / 1	1 / 1	0 / 1	0 / 1	0 / 1	0 / 1	error		error	
d434312d	0 / 1	0 / 2	1 / 2	1 / 2	timeout		error		error	
7beaaeb5	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
4f6bd8de	1 / 1	1 / 2	timeout		2 / 2	2 / 2	2 / 2	2 / 2	2 / 3	2 / 2
633d82c9	1 / 1	1 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2

Table 6. Tool report quality broken down by file.