SHErrLoc: a Static Holistic Error Locator

DANFENG ZHANG, Pennsylvania State University
ANDREW C. MYERS, Cornell University
DIMITRIOS VYTINIOTIS, Microsoft Research Cambridge
SIMON PEYTON-JONES, Microsoft Research Cambridge

18

We introduce a general way to locate programmer mistakes that are detected by static analyses. The program analysis is expressed in a general constraint language which is powerful enough to model type checking, information flow analysis, dataflow analysis and points-to analysis. Mistakes in program analysis result in unsatisfiable constraints. Given an unsatisfiable system of constraints, both satisfiable and unsatisfiable constraints are analyzed, to identify the program expressions most likely to be the cause of unsatisfiability. The likelihood of different error explanations is evaluated under the assumption that the programmer's code is mostly correct, so the simplest explanations are chosen, following Bayesian principles. For analyses that rely on programmer-stated assumptions, the diagnosis also identifies assumptions likely to have been omitted. The new error diagnosis approach has been implemented as a tool called SHErrLoc, which is applied to three very different program analyses, such as type inference for a highly expressive type system implemented by the Glasgow Haskell Compiler (GHC)—including type classes, GADTs, and type families. The effectiveness of the approach is evaluated using previously collected programs containing errors. The results show that when compared to existing compilers and other tools, SHErrLoc consistently identifies the location of programmer errors significantly more accurately, without any language-specific heuristics.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Information flow control*;

Additional Key Words and Phrases: error diagnosis, static program analysis, type inference, information flow, Haskell, OCaml, Jif

ACM Reference format:
DANFENG ZHANG, ANDREW C. MYERS, DIMITRIOS VYTINIOTIS, and SIMON PEYTON-JONES. 2017. SHErrLoc: a Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 18 (August 2017), 47 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Type systems and other static analyses help reduce the need for debugging at run time, but sophisticated type systems and other program analyses can lead to terrible error messages. The

This work was supported by two grant N00014-13-1-0089 from the Office of Naval by MURI grant FA9550-12-1-0400, by two grant from the National Science Foundation (CCF-0964409, CCF-1566411).
Author's addresses: D. Zhang, the bulk of the research was done at Cornell University, and he is currently affiliated with Department of Computer Science and Engineering, Pennsylvania State University; A. Myers, Department of Computer Science, Cornell University; D. Vytiniotis and S. Peyton-Jones, Microsoft Research Cambridge.

0164-0925/2017/8-ART18 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

difficulty of understanding these error messages interferes with the adoption of expressive type systems and other program analyses.

When deep, non-local software properties are being checked, the analysis may detect an inconsistency in a part of the program far from the actual error, resulting in a misleading error message. The problem is that powerful static analyses and advanced type systems reduce an otherwise-high annotation burden by drawing information from many parts of the program. However, when the analysis detects an error, the fact that distant parts of the program influence this determination makes it hard to accurately attribute blame. Determining from an error message where the true error lies can require an unreasonably complete understanding of how the analysis works.

We are motivated to study this problem based on experience with three programming languages: ML, Haskell [35] and Jif [41], a version of Java that statically analyzes the security of information flow within programs. The expressive type systems in all these languages lead confusing, even misleading error messages [28, 55]. Prior work has explored a variety of methods for improving error reporting in each of these languages. Although these methods are usually specialized to a single language and analysis, they still frequently fail to identify the location of programmer mistakes.

In this work, we take a more general approach. The insight is that most program analyses, including type systems and type inference algorithms, can be expressed as systems of constraints over variables. In the case of ML type inference, variables stand for types, constraints are equalities between different type expressions, and type inference succeeds when the corresponding system of constraints is satisfiable. With a sufficiently expressive constraint language, we show that more advanced features in other program analyses, such as programmer' assumption in Jif information flow analysis, quantified propositions involving functions over types, used in GHC, can all be modeled in a concise yet powerful constraint language SCL (Section 4).

SHErrLoc comes with a customized constraint language and solver[1], which identifies both satisfiable and unsatisfiable constraint subsets via a graph representation of the constraint system (Sections 6–8). When constraints are unsatisfiable, the question is how to report the failure indicating an error by the programmer. The standard practice is to report the first failed constraint along with the program point that generated it. Unfortunately, this simple approach often results in misleading error messages—the actual error may be far from that program point. Another approach is to report all expressions that might contribute to the error (e.g., [11, 19, 52, 55]). But such reports are often verbose and hard to understand [24].

Our insight is that when the constraint system is unsatisfiable, a more holistic approach should be taken. Rather than looking at a failed constraint in isolation, the structure of the constraint system as a whole should be considered. The constraint system defines paths along which information propagates; both satisfiable and unsatisfiable paths can help locate the error. An expression involved in many unsatisfiable paths is more likely to be erroneous; an expression that lies on many satisfiable paths is more likely correct. This approach can be justified on Bayesian grounds, under the assumption, captured as a prior distribution, that code is mostly correct (Section 9).

In some languages, the satisfiability of constraint systems depends on environmental assumptions, which we call hypotheses. The same general approach can also be used to identify hypotheses likely to be missing: a small, weak set of hypotheses that makes constraints satisfiable is more likely than a large, strong set.

[1]One downside of using a customized constraint language and solver is the SHErrLoc solver may fall out of sync with the ones in existing compilers, such as GHC. However, this approach is still preferable for its generality, since SHErrLoc is intended to supplement existing compilers when they fail to provide useful error messages. As long as the solvers largely agree on constraints, SHErrLoc can provide meaningful error reports when existing compilers are unsatisfactory.

In summary, this article presents the following contributions:

(1) We define a constraint language, SCL, and its constraint graph representation that can encode a broad range of type systems and other analyses. In particular, we show that SCL can express a broad range of program analyses, such as ML type inference, Jif information flow analysis, many dataflow analyses, points-to analysis and features of the expressive type system of Haskell, including type classes, GADTs, and type families (Section 4 and 5).
(2) We present a novel constraint-graph-based solving technique that handles the expressive SCL constraint language. The novel technique allows the creation of new nodes and edges in the graph and thereby to support counterfactual reasoning about type classes, type families, and their universally quantified axioms. We prove that the new algorithm always terminates (Section 6–8).
(3) We develop a Bayesian model for inferring the most likely cause of program mistakes identified in the constraint analysis. Using a Bayesian posterior distribution [18], the algorithm suggests program expressions that are likely errors and offers hypotheses that the programmer is likely to have omitted (Section 9).
(4) We evaluate the accuracy and performance of SHErrLoc on three different sets of programs written in OCaml, Haskell and Jif. As part of this evaluation, we use large sets of programs collected from students using OCaml and Haskell to do programming assignments [20, 31]. Appealingly, high-quality results do not rely on language-specific tuning (Section 10).

Contributions in relation to prior versions. This article supersedes its previous conference versions presented at the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages [58] and the ACM SIGPLAN conference on Programming Language Design and Implementation [59] in several ways:

- It provides the syntax (Section 4.1), graph construction (Section 6), and graph saturation algorithm (Section 7) for the complete SCL constraint language. Earlier versions have omitted features for simplicity: the POPL'14 paper [58] lacks quantified axioms in hypotheses and functions over constraint elements; the PLDI'15 paper [59] lacks contravariant/invariant constructors, projections, and join and meet operations on constraint elements.
- It provides an end-to-end overview of the core components of SHErrLoc (Section 2.4). The overview includes information that is omitted in the previous versions, such as a detailed discussion on how constraints are generated, how the Bayesian model works, and how errors are reported by SHErrLoc.
- It provides a running example (Section 3) to give an in-depth view of the advanced features of SHErrLoc. The running example is explained throughout the paper.
- It formalizes the entailment rules for the SCL constraint language (Section 4.2).
- It provides more details on the DLM model [40] and its encoding in the constraint language; in particular, it proves that a confidentiality/integrity policy in the DLM model is a constructor on principals with the appropriate variance (Section 5.2).
- It shows that the SCL language is expressive enough to model a nontrivial program analysis: points-to analysis (Section 5.4).
- It proves that our constraint analysis algorithm always terminates (Section 7.5).
- It also proves that "redundant" graph edges provide no extra information for error localization (Section 8.4).
- It describes an efficient search algorithm, based on A*, that searches for the most-likely explanation of program errors. The algorithm is proved to always return optimal solutions (Section 9.2).

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

OCaml Compiler Report:
Line 9, Characters 33-44: This expression has type 'a list but is here used with type unit.

SHErrLoc Report:
Expressions in the source code that appear most likely to be wrong:
print_string "foo" (Line 7, characters 6-24)

Fig. 1. OCaml example. Line 9 is blamed by OCaml compiler for a mistake at line 7.

```
1 fac n = if n == 0 then 1
2         else n * fac (n == 1)
```

GHC Compiler Report:
Line 1 Column 18: No instance for (Num Bool) arising from the literal '0'

SHErrLoc Report:
Expressions in the source code that appear most likely to be wrong:
n == 1 (Line 2, Characters 23-28)

Fig. 2. Haskell example. Line 1 is blamed for a mistake at line 2.

2 APPROACH

Our general approach to diagnosing errors can be illustrated through examples from three languages: ML, Haskell and Jif.

2.1 ML type inference

The power of type inference is that programmers may omit types. But when type inference fails, the resulting error messages can be confusing. Consider Figure 1, containing (simplified) OCaml code written by a student working on a programming assignment [31]. The OCaml compiler reports that the expression [(`0.0`, `0.0`)] at line 9 is a list, but is used with type `unit`. However, the programmer's actual fix shows that the error is the `print_string` expression at line 7.

The misleading report arises because currently prevalent error reporting methods (e.g., in OCaml [43], SML [39], and Haskell [26]) unify types according to type constraints or typing rules, and report the last expression considered, the one on which unification fails. However, the first failed expression can be far from the actual error, since early unification using an erroneous expression may lead type inference down a garden path of incorrect inferences.

In our example, the inference algorithm unifies (i.e., equates) the types of the four highlighted expressions, in a particular order built into the compiler. One of those expressions, [(`0.0`, `0.0`)], is blamed because the inconsistency is detected when unifying its type.

Prior work has attempted to address this problem by reporting either the complete *slice* of the program relating to a type inference failure, or a smaller subset of unsatisfiable constraints [11, 19, 52, 55]. Unfortunately, both variants of this approach can still require considerable manual effort to identify the actual error within the program slice, especially when the slice is large [24].

2.2 Haskell type inference

Haskell is recognized as having a particularly rich type system, and hence makes an excellent test case for our approach. Consider the Haskell program from [30] in Figure 2, which fails to type-check. The actual mistake is that the second equality test (==, in line 2) should be subtraction

```
1 public final byte[{}]{this} encText;
2 ...
3 public void m(FileOutputStream[{this}]{this} encFos)
4     throws (IOException) {
5   try {
6     for (int i=0; i<encText.length; i++)
7       encFos.write(encText[i]);
8   } catch (IOException e) {}
9 }
```

Jif Compiler Report:
Line 3: The non-exception termination of the method body may reveal more information than is declared by the method return label.

SHErrLoc Report:
Expressions in the source code that appear most likely to be wrong:
{} (Line 1, Characters 19-20)

Fig. 3. Jif example. Line 3 is blamed for a mistake at line 1.

(-), but GHC instead blames the literal `0`, saying that **`Bool`** is not a numerical type. A programmer reading this message would probably be confused why `0` should have type **`Bool`**. Unfortunately, such confusing error messages are not uncommon.

The core of the problem is that like ML, GHC implements constraint solving by iteratively simplifying type constraints, making error reporting sensitive to the order of simplification. GHC here decides to first unify the return type of `(n == 1)`, namely **`Bool`**, with the type of `n`, which is the argument of `fac`. Once the type of `n` is fixed to **`Bool`**, the compiler picks up the constraint arising from line 1, (expression `n == 0`), unifies the type of `0` with **`Bool`** and reports misleadingly that literal `0` is the error source.

2.3 Jif label checking

Confusing error messages are not unique to traditional type inference. The analysis of information flow security, which checks a different kind of nonlocal code property, can also generate confusing messages when security cannot be verified.

Jif [41] is a Java-like language whose static analysis of information flow often generates confusing error messages [28]. Figure 3 shows a simplified version of code written by a Jif programmer. Jif programs are similar to Java programs except that they specify security labels, shadowed in the example. A security label describes the intended confidentiality and integrity for the associated data. Omitted labels (such as the label of `i` at line 6) are inferred automatically. However, Jif label inference works differently from ML type inference algorithms: the type checker generates constraints on labels, creating a system of inequalities that are then solved iteratively. For instance, the compiler generates a constraint $\{\} \leq \{\texttt{this}\}$ for line 7, bounding the label of the argument `encText[i]` by that on the formal parameter to `write()`, which is `{this}` because of `encFos`'s type.

Jif error messages are a product of the iterative process used to solve these constraints. The solver uses a two-pass process that involves both raising lower bounds and lowering upper bounds on labels to be solved for. Errors are reported when the lower bound on a label cannot be bounded by its upper bound.

As with ML, early processing of an incorrect constraint may cause the solver to detect an inconsistency later at the wrong location. In this example, Jif reports that a constraint at line 3 is wrong, but the actual programmer mistake is the label {} at line 1.

An unusual feature of Jif is that programmers may specify assumptions, capturing trust relationships that are expected to hold in the environment in which the program is run. A common reason why label checking fails in Jif is that the programmer has gotten these assumptions wrong. Sharing constraints on ML functor parameters are also assumptions, but are simpler and less central to ML programming.

(1) `unit` print_string (7) = α acc (5)
(2) α acc (5) = β acc (3)
(3) β acc (3) = `(float*float) list` `[(0.0,0.0)]`(9)
(4) `unit` print_string (7) = γ loop ret (9)
(5) γ loop ret (9) = δ `list` List.rev (9)
(6) δ `list` List.rev (9) = `(float*float) list` f ret (1)
(7) γ loop ret (9) = α acc (5)

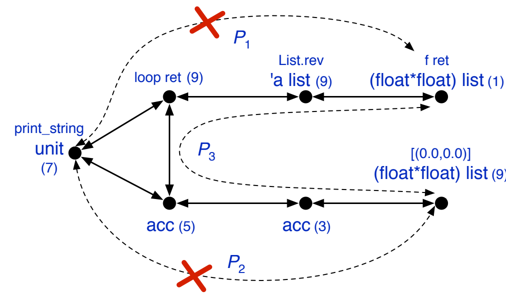


Fig. 4. Part of the constraints generated from the OCaml example in Figure 1 (left) and the corresponding constraint graph (right).

For instance, an assignment from a memory location labeled with a patient's security label to another location with a doctor's label might fail to label-check because the crucial assumption is missing that the doctor acts for the patient. With that assumption in force, it is clear that an information flow from patient to doctor is secure.

In this article, we propose a unified way to infer both program expressions likely to be wrong and assumptions likely to be missing.

2.4 Overview of the approach

We use the OCaml example in Figure 1 to provide an end-to-end overview of SHErrLoc.

Constraints. As a basis for a general way to diagnose errors, we define an expressive constraint language, SCL, that can encode a large class of program analyses, including not only ML, Haskell type inference and Jif label checking, but also dataflow analyses and points-to analysis.

Constraints in this language assert partial orderings on constraint elements, in the form of $E_1 \leq E_2$, where E_1 and E_2 are constraint elements. Each constraint element and constraint is associated with meta data (e.g, the corresponding line number and expression in the source code). They are only used for better readability: SHErrLoc use the meta data to map an identified error cause in the constraint system back into the source code.

For example, the code in Figure 1 generates a constraint system containing several assertions, including but not limited to the ones shown in Figure 4 (here, $E_1 = E_2$ is a short hand for $E_1 \leq E_2 \wedge E_2 \leq E_1$). For simplicity, only the meta data (expression and line number) for each constraint elements are shown in shades. Here, α, β, γ etc. represent type variables to be inferred, while other constraint elements are data types. For example, the first constraint states that the type of the result of `print_string`, which is `unit`, must be identical to the type of the expression `acc` at line 5, since the "then" and "else" branches must have the same type in OCaml. The sixth constraint states that the type of the return value of function `List.rev` must be identical to the type of the return value of `f`, which is `(float*float) list` as provided in the function signature.

Constraint graph. The constraints are then converted into a representation as a directed graph. In that graph, a node represents a constraint element, and a directed edge represents an ordering between the two elements it connects.

For example, the right of Figure 4 (excluding dashed edges) shows the constructed constraint graph for the constraints shown on the left. The leftmost node represents the type of the result of `print_string` (i.e., constraint element `unit`). The leftmost node is connected by edges to the node representing the result type of `loop` due to the constraint `unit` = γ.

Graph saturation. To identify potential conflicts in the constraints, more (shortest) directed paths that must hold are inferred in the graph representation, in a form of reachability: CFL-reachability [49], to be precise. For example, the dashed paths P_1, P_2 and P_3 in Figure 4 can be inferred due to the transitivity of $\leq$.

Type inference fails if there is at least one unsatisfiable path within the constraint graph, indicating a sequence of unifications that generate a contradiction. Consider, for example, the three paths P_1, P_2, and P_3 in the figure. The end nodes of each path must represent the same types. Other such inferred paths exist, such as a path between the node for `unit` and the node `acc` (3), but these paths are not shown since a path with at least one variable on an end node is trivially satisfiable. We call paths that are not trivially satisfiable, such as P_1, P_2, and P_3, the *informative* paths. We note that other informative paths can be inferred, such as a path from `unit` to δ `list` (9); these paths are omitted for simplicity.

In this example, the paths P_1 and P_2 are unsatisfiable because the types at their endpoints are different. Note that path P_2 corresponds to the expressions highlighted in the OCaml code. By contrast, path P_3 is satisfiable.

Bayesian reasoning. The constraints along unsatisfiable paths form a complete explanation of the error, but one that is often too verbose. Our goal is to be more useful by pinpointing where along the path the error occurs. The key insight is to analyze both satisfiable and unsatisfiable informative paths identified in the constraint graph.

In Figure 4, the strongest candidate for the real source of the error is the leftmost node of type `unit`, rather than the lower-right expression of type `(float*float) list` that features in the misleading error report produced by OCaml. Two general heuristics help us identify `unit` as the culprit:

(1) All else equal, an explanation for unsatisfiability in which programmers have made fewer mistakes is more likely. This is an application of Occam's Razor. In this case, the minimum explanation is a single expression (the `unit` node) which appears on both unsatisfiable paths.
(2) Erroneous nodes are less likely to appear in satisfiable paths. In this case, The `unit` node appears only on unsatisfiable informative paths, but not on the informative, satisfiable path P_3. Hence, the `unit` node is a better error explanation than any node lying on path P_3.

We note that in Figure 4, the first heuristic alone already promotes the real source of the error (the leftmost node of type `unit`) as the strongest candidate. However, in general, the second heuristic improves error localization accuracy as well. For example, consider a constraint graph that is identical to Figure 4 except that the bottom left solid edge between `unit` and `acc` (5) is removed. In this graph, P_2 is removed as well. Hence, all nodes along path P_1 are equally likely to be wrong according to the first heuristic. In this case, the second heuristics is needed to identify the leftmost node as the strongest candidate.

A Bayesian model justifies the intuitive heuristics above. To see why, we interpret the error diagnosis process as identifying an *explanation* of observing the saturated constraint graph. In this case, the observation o is the satisfiability of informative paths within the constraint graph. We denote the observation as $o = (o_1, o_2, \ldots, o_n)$, where $o_i \in \{\texttt{unsat}, \texttt{sat}\}$ represents unsatisfiability or satisfiability of the corresponding path. Let $P(E|o)$ be the posterior probability that a set of nodes E explains the given observation o. By Bayes' theorem, maximizing $P(E|o)$ is equivalent to maximizing the term

$$P(E)P(o|E)$$

With a couple of simplifying assumptions (Section 9.1), the most-likely explanation can be identified as a set of nodes, E, such that

(1) Any unsatisfiable path uses at least one node in E, and
(2) E minimizes the term $C_1|E| + C_2k_E$, where k_E is the number of satisfiable paths using at least one node in E, and C_1, C_2 are some tunable constants.

We note that the Bayesian model justifies the intuitive heuristics above: the explanation is likely to contain fewer nodes (heuristic 1) and show less frequently on satisfiable edges (heuristic 2). Appealingly, these two heuristics rely only on graph structure, and are oblivious to the language and program being diagnosed. The same generic approach can therefore be applied to very different program analyses.

Error reporting. SHErrLoc uses an instance of A* search algorithm to identify top-ranked explanations according to the term $C_1|E| + C_2k_E$. Each explanation consists of one or multiple program expressions. For example, SHErrLoc reports the only top-ranked explanation for the OCaml program in Figure 1 as follows:

> Expressions in the source code that appear most likely to be wrong:
> `print_string "foo"` (Line 7, characters 6-24)

This explanation is exactly the true mistake in the program, according to the programmer's actual error fix.

For the programs in Figure 2 and Figure 3, the SHErrLoc reports are shown on the right of the figures. Again, SHErrLoc correctly and precisely localizes the actual causes of the errors in those examples.

3 RUNNING EXAMPLE

To explore more advanced features of SHErrLoc, we use the Haskell program in Figure 5 as a running example for the rest of this article. This example involves a couple of sophisticated features in Haskell:

- **Type classes** introduce, in effect, relations over types, on top of ordinary unification constraints. For example, the type of literal `0` can be any instance of the type class `Num`, such as `Int` and `Float`.

- **Type families** are functions at the level of types:

```
type instance F [a] = (Int,a)
f :: F [Bool] -> Bool
f x = snd x
```

  In this example, it is okay to treat `x` as a pair although it is declared to have type `F [Bool]`, because of the axiom describing the behavior of the type family `F`. (Note that in Haskell, type `[Bool]` represents a list of `Bool`'s.)

- **Type signatures.** Polymorphic type signatures introduce universally quantified variables that cannot be unified with other types [46]. Consider the following program.

```
f :: forall a. a -> (a,a)
f x = (True,x)
```

  This program is ill-typed, as the body of `f` indicates that the function is not really polymorphic (consider applying `f 42`).
  Moreover, it is unsound to equate a type variable bound in an outer scope to a universally quantified variable from an inner scope. Consider the following program.

```
f x = let g :: forall a. a -> (a,a)
```

```
-f::∀c. (c,c) → c
-g::∀d. Num d ⇒ d → Bool
-assume Q = (∀a. F[a]=(a,a))
-                ∧ ([Int] ≤ Num)

let h::∀a b. a=[b] ⇒ (F a) → b
         = λx. f x
in g ['a'] -- error
```

$$(a\colon \mathrm{a}_0, b\colon \mathrm{b}_0, \mathtt{x}\colon \chi_1, c\colon \xi_2, \mathtt{f\ x}\colon \phi_2,$$
$$d\colon \delta_0, \mathtt{'a'}\colon \alpha_0, \mathtt{g\ ['a']}\colon \gamma_0)$$

$$\begin{aligned} & H' \vdash \chi_1 \to \phi_2 = (\xi_2, \xi_2) \to \xi_2 \\ \wedge\ & H' \vdash \chi_1 \to \phi_2 = (F\ \mathrm{a}_0) \to \mathrm{b}_0 \\ \wedge\ & H \vdash \delta_0 \leq \mathrm{Num} \\ \wedge\ & H \vdash \alpha_0 = \mathrm{Char} \\ \wedge\ & H \vdash [\alpha_0] \to \gamma_0 = \delta_0 \to \mathrm{Bool} \end{aligned}$$

where $H = (\forall a\ .\ F\ [a] = (a, a)) \wedge ([\mathrm{Int}] \leq \mathrm{Num})$

$$H' = H \wedge (\mathrm{a}_0 = [\mathrm{b}_0])$$

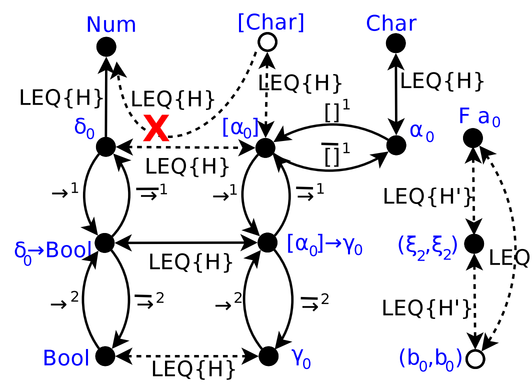


Fig. 5. Running example. Top left: source program; Bottom left: generated constraints; Right: part of the graph for constraints.

```
        g z = (z,x)
    in (g 42, g True)
```

This program is ill-typed, since x's type bound in the enclosing context should not be unified to a, the universally quantified variable from the signature of g. Indeed, if we were to allow this unification, we'd be treating x as having both type **Int** and **Bool** at the two call sites of g. The same issues arise with other GHC extensions, such as data constructors with existential variables and higher-rank types [46].

- **Local hypotheses.** Type signatures with constraint contexts and GADTs both introduce hypotheses under which we must infer or check types. For instance:

```
1 elem :: forall a. Eq a => a->[a]->Bool
2 elem x []     = False
3 elem x (y:ys) = if (x == y) then True
4                 else elem x ys
```

The type signature for **elem** introduces a constraint hypothesis **Eq** a, on the universally quantified variable a, and that constraint is necessary for using == at line 3.

In Figure 5, relevant axiom schemes and function signatures are shown in comments. Here, the type family F maps $[a]$, for an arbitrary type a, to a pair type (a, a). The function h is called only when $a = [b]$. Hence, the type signature is equivalent to $\forall b.\ (b, b) \to b$, so the definition of h is well-typed. On the other hand, expression (g ['a']) has a type error: the parameter type [Char] is not an instance of class Num, as required by the type signature of g.

The informal reasoning above corresponds to a set of constraints, shown on the left bottom of Figure 5. From the constraints, SHErrLoc builds and saturates a constraint graph (shown on

Unification variables	α, β, γ	Constructors	*con*	Quantified variables in hypothesis
Skolem variables	a, b, c	Functions	*fun*	a, b, c

$$G ::= A_1 \wedge \ldots \wedge A_n \;^{(n\geq 0)} \qquad A ::= H \vdash I$$
$$H ::= Q_1 \wedge \ldots \wedge Q_n \;^{(n\geq 0)} \qquad Q ::= \forall \overline{a}\;.\; C \Rightarrow I$$
$$C ::= I_1 \wedge \ldots \wedge I_n \;^{(n\geq 0)} \qquad I ::= E_1 \leq E_2$$
$$E ::= \alpha_\ell \;\mid\; \mathrm{a}_\ell \;\mid\; a \;\mid\; con_{\overline{p}}\;\overline{E} \;\mid\; \overline{con}^{\,i}_{p_i}\;E \;\mid\; fun\;\overline{E} \;\mid\; E_1 \sqcup E_2 \;\mid\; E_1 \sqcap E_2 \;\mid\; \bot \;\mid\; \top$$
$$p ::= + \;\mid\; - \;\mid\; \pm$$

Fig. 6. Syntax of SCL constraints.

the right of Figure 5), where Bayesian reasoning is performed on. We will return to the running example when relevant components are introduced.

4 THE SCL CONSTRAINT LANGUAGE

Central to our approach is a general core constraint language, SCL, that can be used to capture a large class of program analyses. In this constraint language, constraints are inequalities using an ordering $\leq$ that corresponds to a flow of information through a program. The constraint language also has constructors and destructors corresponding to computation on that information, quantified axioms, nested universally and existentially quantified variables, and type-level functions.

4.1 Syntax

The syntax of the SCL (for SHErrLoc Constraint Language) is formalized in Figure 6.

A top-level goal G to be solved is a conjunction of assertions A. An assertion has the form $H \vdash I$, where H is a *hypothesis* (an assumption) and I is an inequality to be checked under H.

Constraints. A constraint C is a possibly empty conjunction of inequalities $E_1 \leq E_2$ over elements from the constraint element domain E (e.g., types of the source language), where $\leq$ defines a partial ordering on elements. Throughout, we write equalities ($E_1 = E_2$) as syntactic sugar for ($E_1 \leq E_2 \wedge E_2 \leq E_1$), and ($H \vdash E_1 = E_2$) is sugar for two assertions, similarly. We denote an empty conjunction as $\emptyset$, and abbreviate $\emptyset \vdash C$ as $\vdash C$.

The ordering $\leq$ is treated abstractly, but when the usual join ($\sqcup$) and meet ($\sqcap$) operators are used in constraints, it must define a lattice. The bottom and top of the element ordering are $\bot$ and $\top$.

Quantified axioms in hypotheses. Hypotheses H can contain (possibly empty) conjunctions of *quantified axioms*, Q. Each axiom has the form $\forall \overline{a}.\ C \Rightarrow I$, where the quantified variables $\overline{a}$ may be used in constraints C and inequality I. For example, a hypothesis $\forall a.\ a \leq A \Rightarrow a \leq B$ states that for any constraint element a such that ($a \leq A$) is valid, inequality $a \leq B$ is valid as well. When both $\overline{a}$ and C are empty, an axiom Q is written simply as I.

Handling quantifiers. To avoid notational clutter associated with quantifiers, we do not use an explicit mixed-prefix quantification notation. Instead, we distinguish universally introduced variables ($\mathrm{a}, \mathrm{b}, \ldots$) and existentially introduced variables ($\alpha, \beta, \ldots$); further, we annotate each variable with its *level*, a number that implicitly represents the scope in which the variable was introduced. For example, we write the formula $\mathrm{a}_1 = \mathrm{b}_1 \vdash (\mathrm{a}_1, \mathrm{b}_1) = \alpha_2$ to represent $\forall \mathrm{a,b}.\exists \alpha.\ \mathrm{a} = \mathrm{b} \vdash$

$(\mathsf{a},\mathsf{b})=\alpha$. Any assertion written using nested quantifiers can be put into prenex normal form [42] and therefore can represented using level numbers.

Constructors and functions over constraint elements. An element E may be 1) a variable, 2) an application $con_{\overline{p}}\,\overline{E}$ of a type constructor $con \in \mathbf{Con}$, where the annotation $\overline{p}$ describes the *variance* of the parameters, or 3) an application $fun\,\overline{E}$ of a type-function $fun \in \mathbf{Fun}$. Constants are nullary constructors, with arity 0. Since constructors and functions are global, they have no levels.

The partial ordering on two applications of the same constructor is determined by the variances $\overline{p}$ of that constructor's arguments. For each argument, the ordering of the applications is either covariant with respect to that argument (denoted by +), contravariant with respect to that argument (−), or invariant (±) with respect to it. For simplicity, we omit the variance p when it is irrelevant to the context.

The main difference between a type constructor *con* and a type function *fun* is that constructors are injective and can be therefore be decomposed (that is, $con\,\overline{\tau} = con\,\overline{\tau'} \Rightarrow \overline{\tau} = \overline{\tau'}$). Type functions are not necessarily injective: $fun\,\overline{\tau}=fun\,\overline{\tau'}$ does not entail that $\overline{\tau}=\overline{\tau'}$.

Example. To model ML type inference, we can represent the type (int → bool) as a constructor application $\mathsf{fn}_{(-,+)}(\mathsf{int},\mathsf{bool})$, where int and bool are constants, the first argument is contravariant, and the second argument is covariant. Its first projection $\overline{\mathsf{fn}}^{1}(\mathsf{fn}_{(-,+)}(\mathsf{int},\mathsf{bool}))$ is int.

Consider the expressions `acc` (line 5) and `print_string` (line 7) in Figure 1. These are branches of an `if` statement, so one assertion is generated to enforce that they have the same type: $\vdash acc_{(5)} \leq \mathsf{unit} \wedge \mathsf{unit} \leq acc_{(5)}$.

Section 5 describes in more detail how assertions are generated for ML, Haskell, Jif, dataflow analysis and points-to analysis.

4.2 Validity and satisfiability

An assertion A is *satisfiable* if there is a *level-respecting* substitution θ for A's free unification variables, such that $\theta[A]$ is *valid*.

A substitution θ is level-respecting if the substitution is well-scoped. More formally, $\forall \alpha_l \in dom(\theta), \mathsf{a}_m \in fvs(\theta[\alpha_l]).\ m \leq l$. For example, an assertion $\mathsf{a}_1=\mathsf{b}_1 \vdash (\mathsf{a}_1=\alpha_2 \wedge \alpha_2=\mathsf{b}_1)$ is satisfiable with substitution $[\alpha_2 \mapsto \mathsf{a}_1]$. But $\vdash \alpha_1=\mathsf{b}_2$ is not satisfiable because the substitution $[\alpha_1 \mapsto \mathsf{b}_2]$ is not level-respecting. The reason is that with explicit quantifiers, the latter would look like $\exists\alpha\forall\mathsf{b}.\ \vdash \alpha=\mathsf{b}$ and it would be ill-scoped to instantiate α with b.

A unification-variable-free assertion $H \vdash I$ is *valid* if I is entailed by H, according to the entailment rules in Figure 7, modulo the least equivalence relation that satisfies the commutativity of the operations $\sqcup$ and $\sqcap$.

The entailment rules are entirely standard. Rule (Axiom) instantiates a (potentially) quantified axiom in the following way: for any substitution θ that maps quantified variables $\overline{\alpha}$ to constraint elements $\overline{E}$, substituted constraints $\theta[C_2]$ are entailed whenever $H \vdash \theta[C_1]$. For example, the following assertion is valid by rule (≤ Ref) and (Axiom) (substitute α with A): $\forall\alpha\ .\ \alpha \leq A \Rightarrow \alpha \leq B \vdash A \leq B$. For the special case when both $\overline{\alpha}$ and C_1 are empty, rule (Axiom) simply entails a constraint already stated in the axioms. For example, $A \leq B \vdash A \leq B$ is (trivially) valid.

The (constructor) composition rule (DComp) checks componentwise relationships according to components' variances; the decomposition rule (DeComp) does the opposite: it infers relationships on components based on their variances.

$$(\leq \textsc{Ref})\ \frac{}{H \vdash E \leq E} \qquad (\leq \textsc{Tran})\ \frac{H \vdash E_1 \leq E_2 \quad H \vdash E_2 \leq E_3}{H \vdash E_1 \leq E_3}$$

$$(\textsc{Axiom})\ \frac{\theta = [\overline{a} \mapsto \overline{E}] \quad H \vdash \theta[C_1]}{H \wedge (\forall \overline{a} \,.\, C_1 \Rightarrow C_2) \vdash \theta[C_2]} \qquad (\textsc{DComp})\ \frac{\begin{array}{l} H \vdash E_i \leq E_i' \ ^{\text{if } p_i=+ \text{ or } p_i=\pm} \\ H \vdash E_i' \leq E_i \ ^{\text{if } p_i=- \text{ or } p_i=\pm} \end{array} \quad 1 \leq i \leq n}{H \vdash con_{\overline{p}}(E_1, \ldots, E_n) \leq con_{\overline{p}}(E_1', \ldots, E_n')}$$

$$(\textsc{Decomp})\ \frac{H \vdash con_{\overline{p}}(E_1, \ldots, E_{a(c)}) \leq con_{\overline{p}}(E_1', \ldots, E_{a(c)}')}{H \vdash \bigwedge_{i=1}^{n} E_i \leq E_i' \ ^{\text{if } p_i=+ \text{ or } p_i=\pm} \wedge E_i' \leq E_i \ ^{\text{if } p_i=- \text{ or } p_i=\pm}}$$

$$(\textsc{FComp})\ \frac{H \vdash E_i \leq E_i' \wedge H \vdash E_i' \leq E_i \qquad 1 \leq i \leq n}{H \vdash \mathit{fun}(E_1, \ldots, E_n) \leq \mathit{fun}(E_1', \ldots, E_n') \wedge \mathit{fun}(E_1', \ldots, E_n') \leq \mathit{fun}(E_1, \ldots, E_n)}$$

$$(\textsc{Join1})\ \frac{H \vdash E_1 \leq E \wedge H \vdash E_2 \leq E}{H \vdash E_1 \sqcup E_2 \leq E} \qquad (\textsc{Join2})\ \frac{}{(H \vdash E_1 \leq E_1 \sqcup E_2) \wedge (H \vdash E_2 \leq E_1 \sqcup E_2)}$$

$$(\textsc{Meet1})\ \frac{H \vdash E \leq E_1 \wedge H \vdash E \leq E_2}{H \vdash E \leq E_1 \sqcap E_2} \qquad (\textsc{Meet2})\ \frac{}{(H \vdash E_1 \sqcap E_2 \leq E_1) \wedge (H \vdash E_1 \sqcap E_2 \leq E_2)}$$

Fig. 7. Entailment rules.

5 EXPRESSIVENESS

The constraint language is the interface between various program analyses and SHErrLoc. To use SHErrLoc, the program analysis implementer must instrument the compiler or analysis to express a given program analysis as a set of constraints in SCL.

As we now show, the constraint language is expressive enough to capture a variety of different program analyses. Of course, the constraint language is not intended to express *all* program analyses, such as analyses that involve arithmetic. We leave incorporating a larger class of analyses into our framework as future work.

5.1 ML type inference

ML type inference maps naturally into constraint solving, since typing rules can be usually be read as equality constraints on type variables. Numerous efforts have been made in this direction (e.g., [2, 19, 24, 37, 56]).

Most of these formalizations are similar, so we discuss how Damas's Algorithm T [12] can be recast into our constraint language, extending the approach of Haack and Wells [19]. We follow that approach since it supports let-polymorphism. Further, our evaluation builds on an implementation of that approach.

For simplicity, we only discuss the subset of ML whose syntax is shown in Figure 8. However, our implementation does support a much larger set of language features, including `match` expressions and user-defined data types.

$$e ::= x \mid n \mid e_1 \ + \ e_2 \mid \texttt{fn}\ x \rightarrow e \ \mid e_1\ e_2 \mid \texttt{let}\ x = e_1 \ \texttt{in}\ e_2$$
$$t ::= \alpha \mid \mathsf{int} \mid t \rightarrow t$$

$$\frac{}{x : \langle [\,]\{x \mapsto \{\alpha_x\}\}, \alpha, \alpha_x = \alpha\rangle} \qquad \frac{}{n : \langle [\,], \alpha, \mathsf{int} = \alpha\rangle}$$

$$\frac{e_1 : \langle \Gamma_1, t_1, C_1\rangle \qquad e_2 : \langle \Gamma_2, t_2, C_2\rangle}{e_1 + e_2 : \langle \Gamma_1 \cup \Gamma_2, \alpha, \mathsf{int} = t_1 \wedge \mathsf{int} = t_2 \wedge \mathsf{int} = \alpha \wedge C_1 \wedge C_2\rangle}$$

$$\frac{e : \langle \Gamma, t, C\rangle \qquad \Gamma(x) = T}{\texttt{fn}\ x \rightarrow e : \langle \Gamma\{x \mapsto \emptyset\}, \alpha, \bigwedge\{\alpha_x = t' \mid t' \in T\} \wedge \alpha = \alpha_x \rightarrow t \wedge C\rangle}$$

$$\frac{e_1 : \langle \Gamma_1, t_1, C_1\rangle \qquad e_2 : \langle \Gamma_2, t_2, C_2\rangle}{e_1\ e_2 : \langle \Gamma_1 \cup \Gamma_2, \alpha, t_1 = t_2 \rightarrow \alpha \wedge C_1 \wedge C_2\rangle}$$

$$\frac{e_1 : \langle \Gamma_1, t_1, C_1\rangle \qquad e_2 : \langle \Gamma_2, t_2, C_2\rangle \qquad \Gamma_2(x) = \{t'_1, \ldots, t'_n\}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \langle \Gamma'_1 \cup \Gamma_2\{x \mapsto \emptyset\}, \alpha, \alpha = t_2 \wedge C \wedge C'_1 \wedge C_2\rangle}$$

where $\langle \Gamma_{1,1}, t_{1,1}, C_{1,1}\rangle \ldots \langle \Gamma_{1,k}, t_{1,k}, C_{1,k}\rangle$, $k = \max(1, n)$, are fresh variants of $\langle \Gamma_1, t_1, C_1\rangle$, $\Gamma'_1 = \bigcup_{1 \leq i \leq k} \Gamma_{1,i}$, $C'_1 = \bigwedge_{1 \leq i \leq k} C_{1,i}$ and $C = \{t_{1,1} = t'_1, \ldots, t_{1,n} = t'_n\}$

Fig. 8. Constraint generation for a subset of ML. α and α_x are fresh variables in typing rules.

In this language subset, expressions can be variables (x), integers (n), binary operations (+), functions abstractions $\texttt{fn}\ x \rightarrow e$, function applications ($e_1\ e_2$), or let bindings ($\texttt{let}\ x = e_1\ \texttt{in}\ e_2$). Notice that let-polymorphism is allowed, such as an expression ($\texttt{let}\ id = \texttt{fn}\ x \rightarrow x\ \texttt{in}\ id\ 2$)

The typing rules that generate constraints are shown in Figure 8. Types t can be type variables to be inferred (α), the predefined integer type int, and function types constructed by $\rightarrow$.

The typing rules have the form $e : \langle \Gamma, t, C\rangle$. Γ is a typing environment that maps a variable x to a set of types. Intuitively, Γ tracks a set of types with which x must be consistent. Let $[\,]$ be an environment that maps all variables to $\emptyset$, and $\Gamma\{x \mapsto T\}$ be a map identical to Γ except for variable x. $\Gamma_1 \cup \Gamma_2$ is a pointwise union for all type variables: $\forall x.(\Gamma_1 \cup \Gamma_2)(x) = \Gamma_1(x) \cup \Gamma_2(x)$. As before, C is a constraint in our language. It captures the type equalities that must be true in order to give e the type t. Note that a type equality $t = t'$ is just a shorthand for the assertion $\vdash t \leq t' \wedge t' \leq t$.

Most of the typing rules are straightforward. To type-check $\texttt{fn}\ x \rightarrow e$, we ensure that the type of x is consistent with all appearances in e, which is done by requiring $\alpha_x = t'$ for all $t' \in \Gamma(T)$. The mapping $\Gamma(x)$ is cleared since x is bound only in the function definition. The rule for let-bindings is more complicated. Because of let-polymorphism, the inferred type of e_1 (t_1) may contain free type variables. To support let-polymorphism, we generate a fresh variant of $\langle \Gamma_1, t_1, C_1\rangle$, where free type variables are replaced by fresh ones, for each use of x in e_2. These fresh variants are then required to be equal to the corresponding uses of x.

Creating one variant for each use in the rule for let-bindings may increase the size of generated constraints, and hence make our error diagnosis algorithm more expensive. However, we find performance is still reasonable with this approach. One way to avoid this limitation is to add polymorphically constrained types, as in [17]. We leave that as future work.

5.2 Information-flow control

In information-flow control systems, information is tagged with security labels, such as "unclassified" or "top secret". Such security labels naturally form a lattice [13], and the goal of such systems is to ensure that all information flows upward in the lattice.

To demonstrate the expressiveness of our core constraint language, we show that it can express the information flow checking in the Jif language [41]. To the best of our knowledge, ours is the first general constraint language expressive enough to model the challenging features of Jif.

Label inference and checking. Jif [41] statically analyzes the security of information flow within programs. All types are annotated with security labels drawn from the decentralized label model (DLM) [40].

Information flow is checked by the Jif compiler using constraint solving. For instance, given an assignment $x := y$, the compiler generates a constraint $L(y) \leq L(x)$, meaning that the label of x must be at least as restrictive as that of y.

The programmer can omit some security labels and let the compiler generate them. For instance, when the label of x is not specified, assignment $x := y$ generates a constraint $L(y) \leq \alpha_x$, where α_x is a label variable to be inferred.

Hence, Jif constraints are broadly similar in structure to our general constraint language. However, some features of Jif are challenging to model.

Label model. The basic building block of the DLM is a set of *principals* representing users and other authority entities. Principals are structured as a lattice with respect to a relation *actsfor*. The proposition *A actsfor B* means *A* is at least as privileged as *B*; that is, *A* is at most as restricted in its activities as *B*.

For instance, if doctor A *actsfor* patient B, then doctor A is allowed to read all information that patient B can read. However, such relation does not grant doctor A to read any information patient C can read, unless doctor A *actsfor* patient C too. The *actsfor* relation can be expressed by the partial ordering $\leq$: for example, the relationship A actsfor B is modeled by the inequality B $\leq$ A.

Security policies on information are expressed as *labels* that mention these principals. For example, the confidentiality label `{patient → doctor}` means that the principal `patient` permits the principal `doctor` to learn the labeled information. Principals can be used to construct integrity labels as well.

A (confidentiality) label L contains a set of principals called the *owners*. For each owner O, the label also contains a set of principals called the *readers*. Readers are the principals to whom owner O is willing to release the information.

For instance, a label $\{o_1 \to r_1 \sqcap r_2;\ o_2 \to r_2 \sqcap r_3\}$ can be read as: principal o_1 allows principals r_1 *or* r_2 to read the tagged information, *and* principal o_2 allows principals r_2 *or* r_3 to read. Only the principals in the *effective reader set*, the intersection of the readers of all owners, may read the information.

In the presence of the *actsfor* relation $\leq$, the effective reader set $readers(p, L)$, the set of principals that p believes should be allowed to read information with label L, is defined as follows:

$$readers(p, o \to r) \triangleq \{q \mid \text{if } p \leq o \text{ then}(o \leq q \text{ or } r \leq q)\}$$

When principal p does not trust o (o does not act for p), the effective reader set is all principals, since p does not credit the policy with any significance. In other words, p has to conservatively assume that the information with the label $o \rightarrow r$ is not protected at all when p does not trust o. Note that though the effective reader set for p is all principals in this case, p is not allowed to read the data if neither $o \leq p$ nor $r \leq p$ holds. The reason is that p is not in the effective reader set of o. As we formalize next, information flow is allowed only if a reader is in the effective reader sets of *all* principals.

Using the definition of effective reader set, the "no more restrictive than" relation $\leq$ on confidentiality policies is formalized as:

$$c \leq d \iff \forall p.\ readers(p, c) \supseteq readers(p, d)$$

For example, consider the following Jif code:

```
int {patient→ ⊤} x;
int y = x;
int {doctor→ ⊤} z;
if (doctor actsfor patient) z = y;
```

The two assignments generate two satisfiable assertions (we use the constructor conf instead of $\rightarrow$ here for clarity):

$$\begin{aligned} & \vdash \mathsf{conf}(\mathsf{patient}, \top) \leq \alpha_y \\ \wedge \qquad \mathsf{patient} \leq \mathsf{doctor} & \vdash \alpha_y \leq \mathsf{conf}(\mathsf{doctor}, \top) \end{aligned}$$

The principals patient and doctor are constants, and the covariant constructor $\mathsf{conf}(p_1, p_2)$ represents confidentiality labels.

Next, we show that encoding a confidential (integrity) policy in Jif as a covariant (contravariant) constructor in SCL is correct. In particular, we prove that a DLM confidentiality policy can be treated as a covariant constructor on principals. Integrity policies are dual to confidentiality policies, so they can be treated as contravariant constructors on principals.

LEMMA 1. *A confidentiality policy in the DLM model is a covariant constructor on principals, and an integrity policy in the DLM model is a contravariant constructor on principals.*

Proof. It is sufficient to show that $a \rightarrow b \sqsubseteq c \rightarrow d \iff a \leq c \wedge b \leq d$ and $a \leftarrow b \sqsubseteq c \leftarrow d \iff c \leq a \wedge d \leq b$.

$\Longrightarrow$: by definition, $readers(a, a \rightarrow b) \supseteq readers(a, c \rightarrow d)$. If $a \nleq c$, then the second part is the entire principal space. This is a contradiction since $\perp \notin readers(a, a \rightarrow b)$. Given $a \leq c$, $d \in readers(a, c \rightarrow d)$. So $d \in readers(a, a \rightarrow b)$. That is, $a \leq d$ or $b \leq d$. In either case, we have $b \leq d$ by noticing that a is an implicit reader of $a \rightarrow b$, or, $b = a \sqcap \ldots \leq a$. The case for integrity policy is the dual of the prove above.

$\Longleftarrow$: consider any principal p. If $p \nleq a$, $readers(p, a \rightarrow b)$ is the entire principal space, hence result is trivial. Otherwise, $p \leq a \leq c$. Hence, sufficient to show that $\{q \mid a \leq q \text{ or } b \leq q\} \supseteq \{q \mid c \leq q \text{ or } d \leq q\}$ which is obvious from assumptions. The case for integrity policy can be proven similarly. □

Label polymorphism. Label polymorphism makes it possible to write reusable code that is not tied to any specific security policy. For instance, consider a function `foo` with the signature `int foo(bool{A→A} b)`. Instead of requiring the parameter `b` to have exactly the label `{A→A}`, the label serves as an upper bound on the label of the actual parameter.

Modeling label polymorphism is straightforward, using hypotheses. The constraint $\mathsf{C}_\mathsf{b} \leq$ `{A → A}` is added to the hypotheses of all constraints generated by the method body, where the constant C_b represents the label of variable `b`.

Method constraints. Methods in Jif may contain "where clauses", explicitly stating constraints assumed to hold true during the execution of the method body. The compiler type-checks the method body under these assumptions and ensures that the assumptions are true at all method call sites. In the constraint language, method constraints are modeled as hypotheses.

5.3 Dataflow analysis

Dataflow analysis is used not only to optimize code but also to check for common errors such as uninitialized variables and unreachable code. Classic instances of dataflow analysis include reaching definitions, live variable analysis and constant propagation.

Aiken [1] showed how to formalize dataflow analysis algorithms as the solution of a set of constraints with equalities over the following elements (a subclass of the more general *set constraints* in [1]):

$$E ::= A_1 \mid \ \ldots \ \mid A_n \mid \alpha \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E$$

where $A_1, \ldots, A_n$ are constants, α is a constraint variable, elements represents sets of constants, and $\cup, \cap, \neg$ are the usual set operators.

Consider live variable analysis. Let $\mathsf{S}_{\mathtt{def}}$ and $\mathsf{S}_{\mathtt{use}}$ be the set of program variables that are defined and used in a statement S, and let $succ(S)$ be the statement executed immediately after S. Two constraints are generated for statement S:

$$\begin{aligned} S_{in} &= \mathsf{S}_{\mathtt{use}} \cup (S_{out} \cap \neg \mathsf{S}_{\mathtt{def}}) \\ S_{out} &= \bigcup_{X \in succ(S)} X_{in} \end{aligned}$$

where S_{in}, S_{out}, X_{in} are constraint variables.

Our constraint language is expressive enough to formalize common dataflow analyses since the constraint language above is nearly a subset of ours: set inclusion is a partial order, and negation can be eliminated by preprocessing in the common case where the number of constants is finite (e.g., $\neg \mathsf{S}_{\mathtt{def}}$ is finite).

5.4 Points-to analysis

Points-to analysis statically computes a set of memory locations that each pointer-valued expression may point to. The analysis is widely used in optimization and other program analyses. Although points-to analysis is commonly used as a component of more complex analyses, such as escape analysis, the fact that a pointer-valued expression points to an unexpected location may lead to confusing analysis results.

We focus on two commonly used flow-insensitive approaches: the equality-based approach of Steensgaard [51] and the subset-based approach of Andersen [3].

One subtlety in formalizing points-to analysis as constraint solving is that a reference can behave both covariantly and contravariantly depending on whether a value is retrieved or set [16]. Mutable reference type τ can be viewed as an abstract data type with two operations: *get:* $unit \rightarrow \tau$ and *set:* $\tau \rightarrow unit$, where τ is covariant in *get*, and contravariant in *set*. To reflect the *get* and *set* operations of mutable references in typing rules, we follow the approach of Foster et al. [16], who use a constructor $\mathtt{ref}_{(+,-)}$ to choose the flow of information. Here, we demonstrate the expressive power of SCL by casting the constraint generation algorithm proposed by Foster et al. [16] to equivalent typing rules generating SCL constraints. However, the use of projections in SCL allows our typing rules to be simpler. The Andersen-style analysis for an imperative language subset is modeled by the following typing rules, where the generated SCL constraints are implicitly accumulated:

$$n : \bot \qquad x : \mathtt{ref}_{(+,-)}(\mathrm{x}, \mathrm{x}) \qquad \frac{e : \tau}{\&e : \mathtt{ref}_{(+,-)}(\tau, \tau)} \qquad \frac{e : \tau}{*e : \overline{\mathtt{ref}}^{1}_{(+,-)}\ \tau}$$

$$\frac{e_1 : \tau_1 \qquad e_2 : \tau_2 \qquad \overline{\mathtt{ref}}^{1}_{(+,-)}\ \tau_2 \le \overline{\mathtt{ref}}^{2}_{(+,-)}\ \tau_1}{e_1 = e_2}$$

Here, constants have no memory locations, hence have the bottom type. The type of a variable is lifted to a pointer type, where each field is the unique constraint constant representing x's memory location. A reference's type says $\&e$ points to something of the type of e. A dereference retrieves the *get* component of the reference to e. The assignment rule asserts that the *get* component of the reference to e_2 is a subset of the *set* component of e_1's reference.

A variable x points to variable y if the relationship $\mathtt{ref}_{(+,-)}(\mathrm{y}, \mathrm{y}) \le \mathrm{x}$ holds. Consider assignments $(x = \&a; y = \&b; *x = y)$. The first assignment generates the following constraint according to the type system:

$$\overline{\mathtt{ref}}^{1}_{(+,-)}\ \big(\mathtt{ref}_{(+,-)}(\mathtt{ref}_{(+,-)}(\mathrm{a}, \mathrm{a}), \mathtt{ref}_{(+,-)}(\mathrm{a}, \mathrm{a}))\big) \le \overline{\mathtt{ref}}^{2}_{(+,-)}\ (\mathtt{ref}_{(+,-)}(\mathrm{x}, \mathrm{x}))$$

This constraint can be fed into a solver for SCL as is. But for scalability, we can first apply a straightforward optimization that collapses constraints containing a consecutive destructor and constructor. Hence, the previous constraint can be simplified to $\mathtt{ref}_{(+,-)}(\mathrm{a}, \mathrm{a}) \le \mathrm{x}$. Similarly, the other two assignments generate two more (simplified) constraints: $\mathtt{ref}_{(+,-)}(\mathrm{b}, \mathrm{b}) \le \mathrm{y}$ and $\mathrm{y} \le \overline{\mathtt{ref}}^{2}_{(+,-)}\ \mathrm{x}$.

Given these three SCL constraints, the points-to analysis already determines that x points to a (from $\mathtt{ref}_{(+,-)}(\mathrm{a}, \mathrm{a}) \le \mathrm{x}$) and y points to b (from $\mathtt{ref}_{(+,-)}(\mathrm{b}, \mathrm{b}) \le \mathrm{y}$). Further, we can infer an extra inequality $\mathtt{ref}_{(+,-)}(\mathrm{b}, \mathrm{b}) \le \mathrm{a}$ (i.e., a points to b) as follows. Since $\mathtt{ref}_{(+,-)}(\mathrm{a}, \mathrm{a}) \le \mathrm{x}$, and the second component of constructor ref is contravariant, we have $\overline{\mathtt{ref}}^{2}_{(+,-)}\ \mathrm{x} \le \mathrm{a}$. Hence, $\mathtt{ref}_{(+,-)}(\mathrm{b}, \mathrm{b}) \le \mathrm{y} \le \overline{\mathtt{ref}}^{2}_{(+,-)}\ \mathrm{x} \le \mathrm{a}$.

Other language features, such as functions and branches, can be handled by similarly as in [16]. Moreover, a Steensgaard-style analysis [51] can be expressed in SCL by converting all generated inequality constraints into equality constraints. For a soundness proof of the constraint generation algorithm, see [16].

Scalability. We observe that the generated constraints fall in an SCL subset that can be solved by an efficient algorithm for the classic all-pairs CFL-reachability problem [38]. This algorithm is part of the graph-based constraint analysis component of SHErrLoc (Section 7.1). Moreover, Melski and Reps [2000] show that CFL-reachability is interconvertible (with the same complexity) with a class of set constraints without union/intersection/negation, the class generated by the set-constraint-based points-to analysis [16]. Hence, a SHErrLoc-based points-to analysis (at least the constraint analysis component) has the same complexity as the set-constraint-based version, which is shown to achieve running times within a small constant factor of a hand-coded flow-insensitive points-to analysis [16]. The scalability of SHErrLoc might be an issue for flow-sensitive points-to analysis on large programs, but we believe most errors relating to points-to analysis can be exposed in the flow-insensitive version. Although SHErrLoc also performs counterfactual reasoning to identify the most-likely error cause, which is absent in the set-constraint-based version, counterfactual reasoning is unlikely to affect the overall scalability, since empirically, constraint analysis is usually the dominant contributor to computation time (Section 10).

Term variables	x, y, z	Type classes	D
Type variables	a, b, c	Type families	F

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \lambda x\,.\,e \mid e_1\;e_2 \\
 & & & \mid \texttt{let}\;x :: \sigma = e_1\;\texttt{in}\;e_2 \\
\text{Constraints} & P & ::= & P_1 \wedge P_2 \mid \tau_1 = \tau_2 \mid D\;\overline{\tau} \\
\text{Signatures} & \sigma & ::= & \forall \overline{a}\,.\,P \Rightarrow \tau \\
\text{Monotypes} & \tau & ::= & a \mid \mathsf{Int} \mid \mathsf{Bool} \mid [\tau] \mid \mathsf{T}\;\overline{\tau} \mid F\;\overline{\tau} \\
\text{Axiom schemes} & Q & ::= & P \mid Q_1 \wedge Q_2 \mid \forall \overline{a}\,.\,P \Rightarrow D\;\overline{\tau} \mid \\
 & & & \forall \overline{a}\,.\,F\;\overline{\tau} = \tau'
\end{array}
$$

Fig. 9. Syntax of a Haskell-like language.

5.5 GHC type inference

Haskell is recognized as having a particularly rich type system, and hence makes an excellent test case for the expressiveness of SCL. We demonstrate this constructively, by giving an algorithm to generate suitable constraints directly from a Haskell-like program. In particular, we show that SCL is expressive enough to encode all Haskell features we introduced in Section 3.

5.5.1 Syntax. Figure 9 gives the syntax for a Haskell-like language. It differs from a vanilla ML language in four significant ways:

- A let-binding has a user-supplied type signatures (σ) that may be polymorphic. For example,
  ```
  let id :: (∀a . a → a) = (λx.x) in ...
  ```
  declares an identity function with a polymorphic type.
- A polymorphic type σ may include constraints (P), which are conjunctions of type equality constraints ($\tau_1 = \tau_2$) and type class constraints ($D\;\overline{\tau}$). Hence, the language supports multi-parameter type classes. The constraints in type signatures are subsumed by SCL, as we see shortly.
- The language supports type families: the syntax of types τ includes type families ($F\;\overline{\tau}$). A type can also be quantified type variables (a) and regular types (Int, Bool, $[\tau]$) that are no different from some arbitrary data constructor T.
- An axiom scheme (Q) is introduced by a Haskell instance declaration, which we omit in the language syntax for simplicity. An axiom scheme can be used to declare relations on types such as type class instances, and type family equations. For example, the following declaration introduces an axiom ($\forall a\,.\,\mathsf{Eq}\;a \Rightarrow \mathsf{Eq}\;[a]$) into the global axiom schemes Q:
  ```
  instance Eq a => Eq [a] where { ... }
  ```

Implicit let-bound polymorphism. One further point of departure from Hindley-Milner (but not GHC) is the lack of let-bound *implicit* generalization. We decided not to address this feature in the present work for two reasons:

(1) Implicit generalization brings no new challenges from a constraint-solving perspective, the focus of this paper,
(2) It keeps our formalization closer to GHC, which departs from implicit generalization anyway [54].

$$\boxed{\text{Constraint translation } [\![P]\!] : C}$$

$$[\![D\ \tau]\!] := \tau \leq \mathsf{D} \qquad [\![D\ \overline{\tau}]\!] := (\mathsf{tup}_n\ \overline{\tau}) \leq \mathsf{D}$$

$$[\![P_1 \wedge P_2]\!] := [\![P_1]\!] \wedge [\![P_2]\!] \qquad [\![\tau_1 = \tau_2]\!] := (\tau_1 = \tau_2)$$

$$\boxed{\text{Type inference rules } H;\Gamma \models_\ell e : \tau \rightsquigarrow G}$$

$$\frac{(v : \forall \overline{a}\ .\ P \Rightarrow \tau) \in \Gamma \qquad \overline{\alpha_\ell} \text{ fresh}}{H;\Gamma \models_\ell v : [\overline{a \mapsto \alpha_\ell}]\tau \rightsquigarrow H \vdash [\overline{a \mapsto \alpha_\ell}][\![P]\!]}\ (\textsc{VarCon})$$

$$\frac{H;\Gamma,(x : \alpha_\ell) \models_{\ell+1} e : \tau_2 \rightsquigarrow G \qquad \alpha_\ell \text{ fresh}}{H;\Gamma \models_\ell \lambda x\ .\ e : \alpha_\ell \rightarrow \tau_2 \rightsquigarrow G}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c} H;\Gamma \models_\ell e_1 : \tau_1 \rightsquigarrow G_1 \\ H;\Gamma \models_\ell e_2 : \tau_2 \rightsquigarrow G_2 \quad \alpha_\ell \text{ fresh} \end{array}}{H;\Gamma \models_\ell e_1\ e_2 : \alpha_\ell \rightsquigarrow G_1 \wedge G_2 \wedge (H \vdash \tau_1 = (\tau_2 \rightarrow \alpha_\ell))}\ (\textsc{App})$$

$$\frac{\begin{array}{cc} H \wedge H';\Gamma \models_{\ell+1} e_1 : \tau_1 \rightsquigarrow G_1 & \sigma = (\forall \overline{a}\ .\ P \Rightarrow \tau) \\ H;\Gamma, x : \sigma \models_\ell e_2 : \tau_2 \rightsquigarrow G_2 & \overline{\mathsf{a}_\ell} \text{ fresh skolems} \\ G' = (H \wedge H' \vdash (\tau_1 = \tau')) & \tau' = [\overline{a \mapsto \mathsf{a}_\ell}]\tau \\ & H' = [\overline{a \mapsto \mathsf{a}_\ell}][\![P]\!] \end{array}}{H;\Gamma \models_\ell \texttt{let}\ x :: \sigma = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow G_1 \wedge G_2 \wedge G'}\ (\textsc{Sig})$$

Fig. 10. Constraint generation.

5.5.2 Constraint generation. Following prior work on constraint-based type inference [44, 47, 53], we formalize type inference as constraint solving, generating SCL constraints using the algorithm in Figure 10.

The constraint-generation rules have the form $H;\Gamma \models_\ell e : \tau \rightsquigarrow G$, read as follows: "given hypotheses H, in the typing environment Γ, we may infer that an expression e has a type τ and generates assertions G". The level ℓ associated with each rule is used to track the scope of unification (existential) and skolem (universal) variables. Here, both H and G follow the syntax of SCL.

Rule (VarCon) instantiates the polymorphic type of a variable or constructor, and emits an instantiated constraint of that type under the propagated hypothesis. Rule (Abs) introduces a new unification variable at the current level, and checks e with an increased level. Rule (App) is straightforward. Rule (Sig) replaces quantified type variables in type signature with fresh skolem variables. Term e_1 is checked under the assumption (H') that the translated constraint in the type signature (P) holds, under the same replacement. The assumption is checked at the uses of x (Rule (VarCon)). The quantifier level is not increased when e_2 is checked, since all unification/skolem variables introduced for e_1 are invisible in e_2.

Constraints are generated for a top-level expression under the global axiom schemes Q, under the translation below.

Type classes. How can we encode Haskell's type classes in SCL constraints? The encoding is shown in Figure 10: we express a class constraint ($D\ \tau$) as an inequality ($\tau \leq \mathsf{D}$), where D is a unique constant for class D. The intuition is that τ is a member of the set of instances of D. For a multi-parameter type class, the same idea applies, except that we use a constructor tup_n to construct a single element from the parameter tuple of length n.

For example, consider a type class Mul with three parameters (the types of two operands and the result of multiplication). The class Mul is the set of all type tuples that match the operators and result types of a multiplication. Under the translation above, $[\![\mathsf{Mul}\ \tau_1\ \tau_2\ \tau_3]\!] = (\mathsf{tup}_3\ \tau_1\ \tau_2\ \tau_3 \leq \mathsf{Mul})$.

Example. Return to the running example in Figure 5. The shaded constraints are generated for the expression (g ['a']) in the following ways. Rule (VarCon) instantiates d in the signature of g at type δ_0, and generates the third constraint (recall that (Num δ_0) is encoded as ($\delta_0 \leq$ Num)). Instantiate the type of character 'a' at type α_0; hence $\alpha_0 = \mathsf{Char}$. Finally, using (App) on the call (g ['a']) generates a fresh type variable γ_0 and the fifth constraint $([\alpha_0] \to \gamma_0) = (\delta_0 \to \mathsf{Bool})$. These three constraints are unsatisfiable, revealing the type error for g ['a']. On the other hand, the first two (satisfiable) constraints are generated for the implementation of function g. The hypotheses of these two constraints contain $\mathrm{a}_0 = [\mathrm{b}_0]$, added by rule (Sig).

5.6 Errors and explanations

Recall that the goal of this work is to diagnose the cause of errors. Therefore we are interested not just in the satisfiability of a set of assertions, but also in finding the best explanation for why they are not satisfiable. Failures can be caused by both incorrect constraints and missing hypotheses.

Incorrect constraints. One cause of unsatisfiability is the existence of incorrect constraints appearing in the conclusions of assertions. Constraints are generated from program expressions, so the presence of an incorrect constraint means the programmer wrote the wrong expression.

Missing hypotheses. A second cause of unsatisfiability is the absence of constraints in the hypothesis. The absence of necessary hypotheses means the programmer omitted needed assumptions.

In our approach, an *explanation* for unsatisfiability may consist of both incorrect constraints and missing hypotheses. To find good explanations, we proceed in two steps. The system of constraints is first converted into a representation as a constraint graph (Section 6). This graph is then saturated (Section 7) and paths are classified as either satisfiable or unsatisfiable (Section 8). The graph is then analyzed using Bayesian principles to identify the explanations most likely to be correct (Section 9).

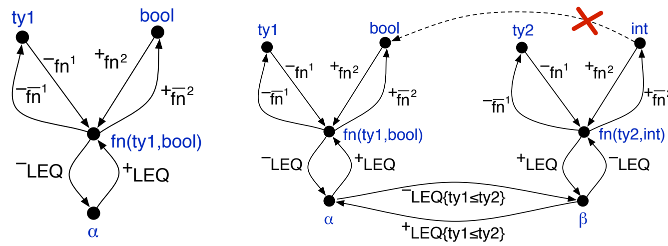
6 CONSTRAINT GRAPH

The SCL language has a natural graph representation that enables analyses of the system of constraints. In particular, the satisfiability of the constraints can be tested via novel algorithms based on context-free-language reachability in the graph.

6.1 Constraint graph construction in a nutshell

A constraint graph is generated from assertions G as follows. As a running example, Figure 5 shows part of the generated constraint graph for the constraints in the center column of the same figure.

(1) For each assertion $H \vdash E_1 \leq E_2$, create *black* nodes for E_1 and E_2 (if they do not already exist), and an edge $\mathsf{LEQ}\{H\}$ between the two. For example, nodes for $\delta_0 \to \mathsf{Bool}$ and $[\alpha_0] \to \gamma_0$ are connected by $\mathsf{LEQ}\{H\}$.
(2) For each constructor node $(con\ \overline{E})$ in the graph, create a *black* node for each of its immediate sub-elements E_i (if they do not already exist); add a labeled *constructor edge* cons^i from the sub-element to the node; and add a labeled *decomposition edge* $\overline{\mathsf{cons}}^i$ in the reverse direction. For example, δ_0 and Bool are connected to $(\delta_0 \to \mathsf{Bool})$ by edges $(\to^1)$ and $(\to^2)$ respectively; and in the reverse direction by edges $\overline{\to}^1$ and $\overline{\to}^2$ respectively.



(a) Constructor edges (b) Full constraint graph

Fig. 11. Detailed constraint graph handling variance.

Repeat step 2 until no more edges or nodes are added.

We address the creation of *dashed edges* and *white nodes* in Sections 7.1 and 7.4 respectively.

6.2 Constructor edge and variance

In general, a constructor edge connecting a constructor node and its components include the following annotations: the constructor name, the argument position, and the *variance* of the parameter (covariant, contravariant or invariant). For simplicity, we omit the variance component for covariant arguments (e.g., in Figure 5), or when the variance is irrelevant in the context.

To illustrate the use of constructor edge to its full extent, we use the following set of constraints:

$$\vdash \alpha \leq \mathsf{fn}_{(-,+)}(\mathtt{ty1}, \mathtt{bool}) \ \wedge \ \mathtt{ty1} \leq \mathtt{ty2} \vdash \beta \leq \alpha \ \wedge \ \vdash \mathsf{fn}_{(-,+)}(\mathtt{ty2}, \mathtt{int}) \leq \beta$$

In this example, we interpret $\leq$ as the subtyping relation. The constructor $\mathsf{fn}_{(-,+)}(E_1, E_2)$ represents the function type $E_1 \longrightarrow E_2$. Note that the constructor fn is contravariant in its first argument and covariant in its second. The identifiers `ty1`, `ty2`, `bool`, `int` are distinct constants and α, β are type variables to be inferred. The constraint graph generated using all three assertions from the example is shown in Figure 11(b), excluding the dashed arrow.

In Figure 11, the edge labeled $({}^{-}\mathsf{fn}^1)$ connects the first (contravariant) argument to the constructor application. As we illustrated in Figure 5, for each constructor edge there is also a dual decomposition edge that connects the constructor application back to its arguments. It is distinguished by an overline above the constructor name in the graph, and has the same variance: for example, $({}^{-}\overline{\mathsf{fn}}^1)$.

To simplify reasoning about the graph with variance, LEQ edges are also duplicated in the reverse direction, with negative variance[2]. Thus, the first assertion in the example, $\vdash \alpha \leq \mathsf{fn}_{(-,+)}(\mathtt{ty1}, \mathtt{bool})$, generates a $({}^{+}\mathsf{LEQ})$ edge from α to $\mathsf{fn}_{(-,+)}(\mathtt{ty1}, \mathtt{bool})$, and a $({}^{-}\mathsf{LEQ})$ edge in the other direction, as illustrated in Figure 11(a).

6.3 Formal construction of the constraint graph

Figure 12 formally presents a function $\mathcal{A}[\![]\!]$ that translates a goal G in SCL (i.e., a set of assertions $A_1 \wedge \ldots \wedge A_n$) into a constraint graph with annotated edges. The graph is represented in the translation as a set of edges defined by the set **Edge**, as well as a set of nodes, defined by the set **Node**, which consists of the legal elements E modulo the least equivalence relation $\sim$ that satisfies the commutativity of the operations $\sqcup$ and $\sqcap$, and that is preserved by the productions in Figure 6.

Most rules are straightforward, but some points are worth noting. First, for each assertion $H \vdash E_1 \leq E_2$, the hypothesis H is merely recorded in the edge labels, to be used by later stages of

[2]This is intentionally omitted in our running example (Figure 5) for readability.

$$
\begin{array}{lll}
n : \textbf{Node} & & (\textbf{Node} = \textbf{Element}/\!\sim) \\
e : \textbf{Edge} & ::= ({}^{p}\mathsf{LEQ})\{H\}(n_1 \mapsto n_2) & \\
& \mid\ ({}^{p_i}\mathit{cons}^i)(n_1 \mapsto n_2)\ \mid\ ({}^{p_i}\overline{\mathit{cons}}^i)(n_1 \mapsto n_2) &
\end{array}
$$

$$
\textbf{Graph} = (\wp(\textbf{Node}), \wp(\textbf{Edge}))
$$

$$
\mathcal{A}[\![G]\!] : \textbf{Graph} \qquad \mathcal{E}[\![E]\!]H : \textbf{Graph}
$$

$$
\begin{aligned}
\mathcal{A}[\![A_1 \wedge \ldots \wedge A_n]\!] &= \bigcup_{i \in 1..n} \mathcal{A}[\![A_i]\!] \\
\mathcal{A}[\![H \vdash E_1 \leq E_2]\!] &= \mathcal{E}[\![E_1]\!]H \cup \mathcal{E}[\![E_2]\!]H \cup\ (\emptyset, \{({}^{+}\mathsf{LEQ})\{H\}(E_1 \mapsto E_2), ({}^{-}\mathsf{LEQ})\{H\}(E_2 \mapsto E_1)\}) \\
\mathcal{E}[\![\alpha_\ell]\!]H &= (\{\alpha_\ell\}, \emptyset) \\
\mathcal{E}[\![\mathsf{a}_\ell]\!]H &= (\{\mathsf{a}_\ell\}, \emptyset) \\
\mathcal{E}[\![\bot]\!]H &= (\{\bot\}, \emptyset) \\
\mathcal{E}[\![\top]\!]H &= (\{\top\}, \emptyset) \\
\mathcal{E}[\![\mathit{con}_{\overline{p}}(\overline{E})]\!]H &= (\{\mathit{con}(\overline{E})\}, \emptyset) \cup \\
&\qquad \textstyle\bigcup_{i \in 1..n} \Big(\mathcal{E}[\![E_i]\!]H\ \cup (\emptyset, \{({}^{p_i}\mathit{con}^i)(E_i \mapsto \mathit{con}(\overline{E})), ({}^{p_i}\overline{\mathit{con}}^i)(\mathit{con}(\overline{E}) \mapsto E_i)\})\Big) \\
\mathcal{E}[\![\overline{\mathit{con}}^i_{p_i}(E)]\!]H &= (\{\overline{\mathit{con}}^i(E)\}, \emptyset) \cup \mathcal{E}[\![E]\!]H \cup (\emptyset, \{({}^{p_i}\mathit{con}^i)(\overline{\mathit{con}}^i(E) \mapsto E), ({}^{p_i}\overline{\mathit{con}}^i)(E \mapsto \overline{\mathit{con}}^i(E))) \\
\mathcal{E}[\![\mathit{fun}(\overline{E})]\!]H &= (\{\mathit{fun}(\overline{E})\}, \emptyset) \cup \bigcup_{i \in 1..n} \mathcal{E}[\![E_i]\!]H \\
\mathcal{E}[\![E_1 \sqcup E_2]\!]H &= (\{E_1 \sqcup E_2\}, \emptyset) \cup \\
&\qquad \textstyle\bigcup_{i \in 1..2} \Big(\mathcal{E}[\![E_i]\!]H\ \cup (\emptyset, \{({}^{+}\mathsf{LEQ})\{H\}(E_i \mapsto E_1 \sqcup E_2), ({}^{-}\mathsf{LEQ})\{H\}(E_1 \sqcup E_2 \mapsto E_i)\})\Big) \\
\mathcal{E}[\![E_1 \sqcap E_2]\!]H &= (\{E_1 \sqcap E_2\}, \emptyset) \cup \\
&\qquad \textstyle\bigcup_{i \in 1..2} \Big(\mathcal{E}[\![E_i]\!]H\ \cup (\emptyset, \{({}^{+}\mathsf{LEQ})\{H\}(E_1 \sqcap E_2 \mapsto E_i), ({}^{-}\mathsf{LEQ})\{H\}(E_i \mapsto E_1 \sqcap E_2)\})\Big)
\end{aligned}
$$

Fig. 12. Construction of the constraint graph

constraint analysis (Section 8). Second, while components of a *constructor* application are connected to the application by constructor/decomposition edges, neither of these edges are added for *function* applications, because function applications cannot be decomposed: $(\mathit{fun}\ A = \mathit{fun}\ B) \not\Rightarrow A = B$.

Third, constructor edges are generated by the rules $\mathcal{E}[\![\mathit{con}_{\overline{p}}(\overline{E})]\!]H$ and $\mathcal{E}[\![\overline{\mathit{con}}^i_{p_i}(E)]\!]H$, which connect a constructor application to its arguments with proper variance annotated on the constructed edges. Invariant arguments generate edges as though they were both covariant and contravariant, so twice as many edges are generated.

Example. Figure 5 (excluding the white nodes, the dashed edges and nodes $F\ [a]$, (ξ_2, ξ_2)) shows the constructed constraint graph for the three shaded constraints on the bottom left of the same figure. For simplicity, edges for reasoning about variance are omitted. Here, the edge from δ_0 to Num is generated from the constraint $H \vdash \delta_0 \leq \mathsf{Num}$, according to the rule for $\leq$. Bi-directional edges between α_0 and Char are generated from the constraint $H \vdash \alpha_0 = \mathsf{Char}$. The rest of the graph is generated from the constraint $H \vdash [\alpha_0] \rightarrow \gamma_0 = \delta_0 \rightarrow \mathsf{Bool}$. Note that according to the rule for constructors, the sub-elements of a constructor are connected to the constructor node with constructor edges. For example, the edges between $[\alpha_0]$ and α_0 as well as the ones between $[\alpha_0] \rightarrow \gamma_0$ and $[\alpha_0]$ are all introduced by the constraint element $[\alpha_0] \rightarrow \gamma_0$.

$$({}^{p}\mathsf{LEQ}\{H_1 \wedge H_2\}) ::= ({}^{p}\mathsf{LEQ}\{H_1\})\ ({}^{p}\mathsf{LEQ}\{H_2\})$$
$$({}^{+}\mathsf{LEQ}\{H\}) ::= ({}^{p}\mathit{con}^{i})\ ({}^{p}\mathsf{LEQ}\{H\})\ ({}^{p}\overline{\mathit{con}}^{i})$$
$$({}^{-}\mathsf{LEQ}\{H\}) ::= ({}^{p}\mathit{con}^{i})\ ({}^{\overline{p}}\mathsf{LEQ}\{H\})\ ({}^{p}\overline{\mathit{con}}^{i})$$
$$\mathsf{LEQ}\{H\}(\mathit{con}(\overline{n}) \mapsto \mathit{con}(\overline{n'})) ::= \mathsf{LEQ}\{H\}(n_i \mapsto n'_i), \forall\, 1 \le i \le |\overline{n}|$$
$$\mathsf{LEQ}\{H\}(\mathit{fun}(\overline{n}) \leftrightarrow \mathit{fun}(\overline{n'})) ::= \mathsf{LEQ}\{H\}(n_i \leftrightarrow n'_i), \forall\, 1 \le i \le |\overline{n}|$$

where $\mathit{con} \in \mathbf{Con}$, $\mathit{fun} \in \mathbf{Fun}$, $p \in \{+,-\}$, $\overline{+} = -$ and $\overline{-} = +$. First two rules apply for consecutive edges.

Fig. 13. Context-free grammar for ($^{+}$LEQ) inference. New edges (left) are inferred based on existing edges (right).

7 GRAPH SATURATION

The key ingredient of graph-based constraint analysis is *graph saturation*: inequalities that are derivable from a constraint system are added as new edges in the graph. We first describe a basic algorithm for constraints where the hypotheses are simply inequalities (i.e., assume $A ::= I$ in Figure 6). Next, we discuss the challenge of analyzing the complete SCL constraints, and then propose a novel algorithm that tackles these challenges.

7.1 Inferring node orderings for simple hypothesis

The basic idea of graph saturation is to construct a context-free grammar, shown in Figure 13, whose productions correspond to inference rules for "≤" relationships.

To perform inference, each production is interpreted as a reduction rule that replaces the right-hand side with the single LEQ edge appearing on the left-hand side. For instance, the transitivity of $\le$ is expressed by the first grammar production, which derives (^{p}LEQ$\{H_1 \wedge H_2\}$) from consecutive LEQ edges (^{p}LEQ$\{H_1\}$) and (^{p}LEQ$\{H_2\}$), where p is some variance. The inferred LEQ edge has hypotheses H_1 and H_2 since the inferred partial ordering is valid only when both H_1 and H_2 hold.

The power of context-free grammars is needed in order to handle reasoning about constructors. In our example using variance (Figure 11), applying transitivity to the constraints yields `ty1` $\le$ `ty2` $\vdash$ `fn(ty2, int)` $\le$ `fn(ty1, bool)`. Then, because `fn` is contravariant in its first argument, we derive `ty1` $\le$ `ty2`. Similarly, we can derive `int` $\le$ `bool`, the dashed arrow in Figure 11(b).

To capture this kind of reasoning, we use the first two productions in Figure 13. In our example of Figure 11(b), the path from `ty1` to `ty2` has the following edges: ($^{-}\mathsf{fn}^{1}$) ($^{-}$LEQ) ($^{-}$LEQ{`ty1` $\le$ `ty2`}) ($^{-}$LEQ) ($^{-}\overline{\mathsf{fn}}^{1}$). These edges reduce via the first and then the second production to an edge ($^{+}$LEQ{`ty1` $\le$ `ty2`}) from `ty1` to `ty2`. Note that the variance is flipped because the first constructor argument is contravariant. Similarly, we can infer another ($^{+}$LEQ{`ty1` $\le$ `ty2`}) edge from `int` to `bool`.

The third grammar production in Figure 13 is the dual of the second production, ensuring the invariant that each ($^{+}$LEQ) edge has an inverse ($^{-}$LEQ) edge. In our example of Figure 11(b), there is also an edge ($^{-}$LEQ{`ty1` $\le$ `ty2`}) from `ty2` to `ty1`, derived from the following edges: ($^{-}\mathsf{fn}^{1}$) ($^{+}$LEQ) ($^{+}$LEQ{`ty1` $\le$ `ty2`}) ($^{+}$LEQ) ($^{-}\overline{\mathsf{fn}}^{1}$). These edges reduce via the first and then the third production to an edge ($^{-}$LEQ{`ty1` $\le$ `ty2`}) from `ty2` to `ty1`.

The last rule applies for function applications, reflecting the entailment rule (FComp) in Figure 7.

Computing all inferable ($^{+}$LEQ) edges according to the context-free grammar in Figure 13 is an instance of *context-free-language reachability*, which is well-studied in the literature [6, 38] and has been used for a number of program-analysis applications [49]. We adapt the dynamic programming algorithm of Barrett et al. [6] to find shortest ($^{+}$LEQ) paths. We call such paths *supporting paths* since the hypotheses along these paths justify the inferred ($^{+}$LEQ) edges. We extend this algorithm to also handle join and meet nodes.

Take join nodes, for instance (meet is handled dually). The rule (JOIN2) in Figure 7 is already handled when we construct edges for join elements (Figure 12).

To handle the rule (JOIN1), we use the following procedure when a new edge $(^{+}\mathsf{LEQ})\{H\}(n_1 \mapsto n_2)$ is processed: for each join element E where n_1 is an argument of the $\sqcup$ operator, we add an edge from E to n_2 if all arguments of the $\sqcup$ operator have a ($^{+}$LEQ) edge to n_2.

7.2 Limitations of pure CFL-reachability analysis

However, graph saturation as described so far is insufficient to handle the full SCL language. We can see this by considering the constraint graph of the running example, in Figure 5. Excluding the white nodes and the edges leading to and from them, this graph is fully saturated according to the rules in Figure 13. For example, the dashed edges between δ_0 and $[\alpha_0]$ can be derived by the second production. However, a crucial inequality (edge) is missing in the saturated graph: ($[\mathsf{Char}] \leq \mathsf{Num}$), which can be derived from the shaded constraints in Figure 5. Since this inequality reveals an error in the program being analyzed (that [Char] is not an instance of class Num), failure to identify it means an error is missed. Moreover, the edges between (ξ_2, ξ_2) and $(F\ \mathsf{a}_0)$ are mistakenly judged as unsatisfiable, as we explain in Section 8.1.

7.3 Expanding the graph

The key insight for fixing the aforementioned problems is to *expand* the constraint graph during graph saturation. Informally, nodes are added to the constraint graph so that the fourth and fifth rules in Figure 13 can be applied.

The (full) constraint graph in Figure 5 is part of the final constraint graph after running our complete algorithm. The algorithm expands the original constraint graph with a new node [Char]. Then, the dashed edge from [Char] to $[\alpha_0]$ is added by the fourth production in Figure 13, and then the dashed edge from [Char] to Num by the first production. Therefore, the unsatisfiable inequality ($[\mathsf{Char}] \leq \mathsf{Num}$) is correctly identified by the complete algorithm. Moreover, the same mechanism determines that $(F\ \mathsf{a}_0) = (\mathsf{b}_0, \mathsf{b}_0)$ can be entailed from hypothesis H', as we explain in Section 8. Hence, edges from and to $(F\ \mathsf{a}_0)$ are correctly classified as satisfiable.

The key challenge for the expansion algorithm is to explore useful nodes without creating the possibility of nontermination. For example, starting from $\alpha_0 = \mathsf{Char}$, a naive expansion algorithm based on the insight above might apply the list constructor to add nodes `[`α_0`]`, `[Char]`, `[[`α_0`]]`, `[[Char]]` and so on infinitely.

7.4 The complete algorithm

To ensure termination, the algorithm distinguishes two kinds of graph nodes: *black nodes* are constructed directly from the system of constraints (i.e., nodes added by the rules in Figure 12); *white nodes* are added during graph expansion.

The algorithm is shown in Figure 14. The top-level procedure `expand&saturate` first initializes the trace for each black node, and then fully expands and saturates a constraint graph. The procedure `saturate` adds (only) new edges to the constraint graph G by using the rules shown in Figure 13.

The most interesting part is the procedure `expand`, which actively adds (only) new white nodes to the graph, so the saturation procedure may saturate the graph further. As depicted in Figure 15, this procedure looks for an LEQ edge between some elements E and E' in the graph G. If G contains

Trace : (**Node**, **Subst**, . . . , **Subst**) **Subst** : (**Element** $\hookrightarrow$ **Element**)

Procedure `expand&saturate`(G : ***Graph***)
  foreach *Element E in G* **do** initialize $\mathcal{T}(E)$ with $(E, \emptyset)$
  call `saturate`(G) and `expand`($G, \mathcal{T}$) until G is unmodified

Procedure `saturate`(G : ***Graph***)
  Add new edges to G according to the rules in Figure 13

Procedure `expand`(G : ***Graph***, $\mathcal{T}$: ***Element*** $\rightarrow$ ***Trace***)
  For a matched pattern shown in Figure 15, say $E_{\texttt{old}}$ is in G already.
  Add $E_{\texttt{new}}$ to G as a white node. Let $E \leq E'$ be an edge between
  the corresponding sub-elements of $E_{\texttt{old}}$ and $E_{\texttt{new}}$:
1 **if** $(E \hookrightarrow E') \notin \mathcal{T}(E_{\texttt{old}})$ **then**
2   initialize $\mathcal{T}(E_{\texttt{new}})$ with (append $(\mathcal{T}(E_{\texttt{old}}),(E \hookrightarrow E'))$)
  end

Fig. 14. Graph saturation and expansion algorithm.

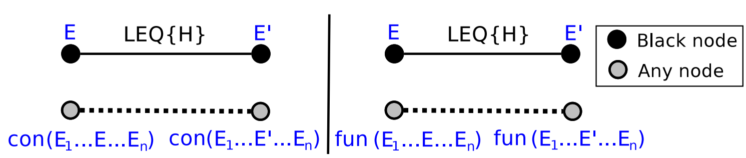


Fig. 15. Graph-expanding patterns. If only one gray node is in the graph, the other one is added as a white node.

only one of *con* $(E_1, ., E, ., E_n)$ and *con* $(E_1, ., E', ., E_n)$, the other element is added as a white node. A similar procedure applies to function applications as well. The added nodes enable more edges to be added by procedure `saturate` (e.g., the dashed edges in Figure 15).

To ensure termination, the expansion procedure places two restrictions on the edges and nodes that trigger expansion. First, both of E and E' must be black nodes. Second, a trace $\mathcal{T}$ is kept for each element. A trace is a single black node along with a sequence of substitutions in the form (**Element** $\hookrightarrow$ **Element**). Intuitively, a trace records how a constraint element can be derived by applying substitutions to an element from the original constraint system (a black node). For example, $((x, y), (x \hookrightarrow \mathsf{Int}), (y \hookrightarrow \mathsf{Bool}))$ is a possible trace for constraint element $(\mathsf{Int}, \mathsf{Bool})$. For a black node, the sequence only contains the node itself. It is required that a single substitution cannot be applied twice (line 1). When a white node is added, a substitution $(E \hookrightarrow E')$ is appended to the trace of $\mathcal{T}(E_{\texttt{old}})$ (line 2).

Returning to our running example in Figure 5, the LEQ edge from α_0 to Char, as well as the node $[\alpha_0]$, match the pattern in Figure 15. In this example, the white node [Char] is added to the graph. As an optimization, no constructor/decomposition edges are added, since these edges are only useful for finding $\alpha_0 = \mathsf{Char}$, which is in the graph already. Moreover, $\mathcal{T}([\mathsf{Char}]) = ([\alpha_0], (\alpha_0 \hookrightarrow \mathsf{Char}))$.

7.5 Termination

The algorithm in Figure 14 always terminates, because the number of nodes in the fully expanded and saturated graph must be finite. This is easily shown by observing that $|\mathcal{T}(E_{\texttt{new}})| = |\mathcal{T}(E_{\texttt{old}})| + 1$, and trace size is finite (elements in a substitution must be black). Here is a formal proof of this fact.

Lemma 2. *The algorithm in Figure 14 always terminates.*

$$\mathbf{HGraph} = (\mathbf{Graph}, \mathcal{R}) \quad \mathcal{R} = \wp(Q) \qquad\qquad \mathcal{Q}[\![H]\!] : \mathbf{HGraph}$$
$$\mathcal{N} : \text{the constraint graph w/o edges} \qquad\qquad \mathcal{A}[\![G]\!] : \text{as defined in Figure 13}$$
$$\mathcal{Q}[\![Q_1 \wedge \ldots \wedge Q_n]\!] = (\mathcal{N}, \emptyset) \cup \bigcup_{i \in 1..n} \mathcal{Q}[\![Q_i]\!]$$
$$\mathcal{Q}[\![I]\!] = (\mathcal{A}[\![\emptyset \vdash I]\!], \emptyset) \qquad \mathcal{Q}[\![\forall \overline{a}\,.\, C \Rightarrow I]\!] = (\emptyset, \{\forall \overline{a}\,.\, C \Rightarrow I\})$$

Fig. 16. Construction of the hypothesis graph.

Proof. We only need to prove that the number of nodes in the fully expanded and saturated graph is finite. To prove this, we notice that the algorithm in Figure 14 maintains an important invariant: $|\mathcal{T}(E)| = |\mathcal{T}(E')| + 1$ where E is added. This is true because $\mathcal{T}(E) = \mathcal{T}(E') \cup \{\mathsf{LEQ}\{H\}(E_1 \mapsto E_2)\}$ (line 2) and the recursion check at line 1.

Therefore, let N be the number of black nodes, SE be the number of edges whose both end nodes are black, and size_i be the number of graph nodes whose trace size is exactly i. It is easy to show $\mathsf{size}_i \leq N \times \mathsf{SE}^{i-1}$ by induction. Moreover, for any element E, $|\mathcal{T}(E)| \leq \mathsf{SE}+1$ because $\mathcal{T}(E)$ may only contain substitutions arising from edges whose both end nodes are black. So $\mathsf{size}_i = 0$ when $i > \mathsf{SE}$. Hence, node size is finite. □

8 CLASSIFICATION

Each LEQ edge $\mathsf{LEQ}\{H\}(E_1 \mapsto E_2)$ in the saturated constraint graph corresponds to an entailment constraint, $H \vdash E_1 \leq E_2$, that is derivable from the constraints being analyzed. For example, in Figure 5, the LEQ edge from $(\mathsf{b}_0, \mathsf{b}_0)$ to $(F\ \mathsf{a}_0)$ corresponds to the following entailment:

$$\begin{array}{l}(\forall a.\ F\ [a] = (a, a)) \wedge \\ ([\mathsf{Int}] \leq \mathsf{Num}) \wedge (\mathsf{a}_0 = [\mathsf{b}_0])\end{array} \vdash (\mathsf{b}_0, \mathsf{b}_0) \leq F\ \mathsf{a}_0$$

Now, the question is: *is this entailment satisfiable*?

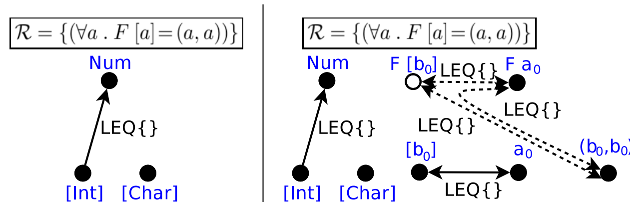
To answer this question, SHErrLoc builds and saturates *hypothesis graphs* for the hypotheses recorded on the LEQ edges. The idea is to infer derivable inequalities from H, so that the satisfiability of $E_1 \leq E_2$ can be simply judged by its existence in the hypothesis graph. Although hypothesis graphs share some similarities with the constraint graph, we note that hypothesis graphs are separate graphs, so building and saturating them does not affect the constraint graph.

8.1 Hypothesis graph

For each hypothesis H shown on LEQ edges in the saturated constraint graph, we construct and saturate a hypothesis graph so that derivable inequalities from H become present in the saturated hypothesis graph.

The construction of a hypothesis graph is shown in Figure 16. For an entailment $H \vdash E_1 \leq E_2$, the constructed graph of H includes both E_1 and E_2. These nodes are needed as guidance for graph saturation. Otherwise, consider an assertion $\mathsf{a}_0 = \mathsf{b}_0 \vdash [[\mathsf{a}_0]] = [[\mathsf{b}_0]]$. Without nodes $[[\mathsf{a}_0]]$ and $[[\mathsf{b}_0]]$, we face a dilemma: either we need to infer (infinite) inequalities derivable from $\mathsf{a}_0 = \mathsf{b}_0$, or we may miss a valid entailment. As an optimization, all nodes (but not edges) in the constraint graph ($\mathcal{N}$) are added to the constructed graph as well. The benefit is that we need to saturate a hypothesis graph just once for all edges that share the hypothesis graph.

The function $\mathcal{Q}[\![H]\!]$ translates a hypothesis H into a graph representation associated with a rule set $\mathcal{R}$. Hypotheses in the degenerate form (I) are added directly; others are added to the rule set $\mathcal{R}$, which is part of a hypothesis graph. Returning to our running example, Figure 17 (excluding the



(a) Hypothesis graph for H in Figure 5.

(b) Hypothesis graph for H' in Figure 5.

Fig. 17. Hypothesis graphs for the running example.

Procedure `saturate`(G : ***Graph***)

1 Add new edges to G according to the rules in Figure 13

2 **foreach** $H = (\forall \overline{a} \,.\, I_1 \wedge \ldots \wedge I_n \Rightarrow E_1 \leq E_2) \in \mathcal{R}$ **do**

3   **if** $\exists \theta : \overline{a} \mapsto \overline{\textit{Node}} \,.\, \forall 1 \leq i \leq n \,.\, \theta[I_i] \in G$ **then**

4   **if** *$\theta[E_1]$ and $\theta[E_2]$ are both in G* **then**

5   add edge from E_1 to E_2 if not in G already

        end

    end

end

Fig. 18. Hypothesis graph saturation for axioms.

white node and dashed edges) shows (part of) the constructed hypothesis graphs for hypotheses H and H'.

The hypothesis graph is then expanded and saturated similarly to the constraint graph. The difference is that axioms are applied during saturation, as shown in Figure 18. In other words, the new algorithm supercedes that in Figure 14 in the extra capability of handling axioms, which are absent in the constraint graph. At line 3, an axiom $\forall \overline{a}.\ C \Rightarrow I$ is applied when it can be instantiated so that all inequalities in C are in G already (i.e., H entails these inequalities). Then, an edge corresponding to the inequality in conclusion is added to G (line 5).

Consider the hypothesis graph in Figure 17(b). The node $F\ [\mathrm{b}_0]$ is added by expand in Figure 14. Moreover, the quantified axiom $(\forall a \,.\, F\ [a] = (a, a))$ is applied, under the substitution $(a \mapsto \mathrm{b}_0)$. Hence, the algorithm adds the dashed edges between $F\ [\mathrm{b}_0]$ and $(\mathrm{b}_0, \mathrm{b}_0)$ to the hypothesis graph. The final saturated hypothesis graph contains edges between $F\ \mathrm{a}_0$ and $(\mathrm{b}_0, \mathrm{b}_0)$ as well, by transitivity. Notice that without graph expansion, this relationship will not be identified in the hypothesis graph, so the edges from and to $(F\ \mathrm{a}_0)$ in Figure 5 are mistakenly classified as unsatisfiable.

8.2 Classification

An entailment $H \vdash E_1 \leq E_2$ is classified as satisfiable iff there is a level-respecting substitution θ such that the hypothesis graph for H contains an LEQ edge from $\theta[E_1]$ to $\theta[E_2]$. Such substitutions are searched for in the fully expanded and saturated hypothesis graph.

Returning to the running example in Figure 5, our algorithm correctly classifies the LEQ edges between $(\mathrm{b}_0, \mathrm{b}_0)$ and $(F\ \mathrm{a}_0)$ as satisfiable, since the corresponding edges are in Figure 17(b). Our algorithm correctly classifies LEQ edges between (ξ_2, ξ_2) and $(F\ \mathrm{a}_0)$ as satisfiable as well, with substitution $\xi_2 \mapsto \mathrm{b}_0$. On the other hand, the LEQ edge from [Char] to Num is (correctly) judged as unsatisfiable, since the inequality is not present in the fully expanded and saturated hypothesis graph for H.

To see why the level-respecting substitution requirement is needed, consider the following example, adapted slightly from the introduction:

```
(λx. let g::(∀a . a → (a, a)) =
         λy. (x, y) in ...)
```

This program generates an assertion $\emptyset \vdash (\beta_2 \longrightarrow (\alpha_0, \beta_2)) = (\mathsf{a}_1 \longrightarrow (\mathsf{a}_1, \mathsf{a}_1))$, which requires that the inferred type for the implementation of g be equivalent to its signature. The final constraint graph for the assertion contains two LEQ edges between nodes α_0 and a_1. These edges are correctly classified as unsatisfiable, since the only substitution candidate, $\alpha_0 \mapsto \mathsf{a}_1$, is not level-respecting.

If the signature of g is $(\forall a \,.\, a = \mathsf{Int} \Rightarrow a \rightarrow (a, a))$, the program is well-typed, since the parameter of g must be Int. This program generates the same assertion as the previous example, but with a hypothesis $\mathsf{a}_1 = \mathsf{Int}$. This assertion is correctly classified as satisfiable, via a level-respecting substitution $\alpha_0 \mapsto \mathsf{Int}$.

8.3 Informative paths

When either end node of a satisfiable ($^+$LEQ) edge is a unification variable, its satisfiability is trivial and hence not informative for error diagnosis. Moreover, when either end node of an ($^+$LEQ) edge is a $\sqcup$ ($\sqcap$) node where at least one argument of $\sqcup$ ($\sqcap$) is a variable, the edge is trivially satisfiable too. For simplicity, we ignore such edges and refer subsequently only to *informative* ($^+$LEQ) edges.

When the partial ordering on the end nodes of a path is invalid, we say that the path is *end-to-end unsatisfiable*. End-to-end unsatisfiable paths are helpful because the constraints along the path explain why the inconsistency occurs.

Also useful for error diagnosis is the set of *satisfiable* paths: paths where there is a valid partial ordering on any two nodes on the path for which a ($^+$LEQ) relationship can be inferred.

Any remaining paths are ignored in our error diagnosis algorithm, since by definition they must contain at least one end-to-end unsatisfiable subpath. For brevity, we subsequently use the term *unsatisfiable path* to mean a path that is *end-to-end* unsatisfiable.

8.4 Redundant edges

The introduction of white nodes introduces *redundant edges*, whose satisfiability is determined by other edges in the graph. Consider Figure 5, the satisfiability of the edge between $[\alpha_0]$ and [Char] merely repeats the edge between α_0 and Char; the fact that end-nodes can be decomposed is also uninformative because white nodes are constructed this way. In other words, this edge provides neither positive nor negative evidence that the constraints it captures are correct. It is *redundant*. We can soundly capture a large class of redundant edges:

Definition 1. *An edge is* redundant *if*

(1) *both end-nodes are constructor applications of the same constructor, and at least one node is white; or*
(2) *both end-nodes are function applications to the same function, and for each simple edge along this edge, at least one of its end-nodes is white.*

Otherwise, an edge is non-redundant.

The following lemma shows that if an edge is redundant according to the previous definition then it does not add any positive or negative information in the graph—it is equivalent to some other set of non-redundant edges.

Lemma 3. *For any redundant edge from E_1 to E_2, there exist non-redundant edges say P_i from E_{i1} to E_{i2}, so that $E_{11} \leq E_{12} \wedge \ldots \wedge E_{n1} \leq E_{n2} \Leftrightarrow E_1 \leq E_2$.*

We first prove some auxiliary results.

Lemma 4 (Edge Simulation). *For any single edge $\mathsf{LEQ}(E \mapsto E')$ in a fully saturated and expanded graph G. If at least one node is white, and*

(1) $E = \mathsf{con}\ \overline{\tau}$ *and* $E' = \mathsf{con}\ \overline{\tau'}$ *for some constructor* con, *or*
(2) $E = \mathsf{fun}\ \overline{\tau}$ *and* $E' = \mathsf{fun}\ \overline{\tau'}$ *for some function* fun,

then for any pair of corresponding parameters τ_i and τ'_i, either $\tau_i = \tau'_i$, or there is an edge $\mathsf{LEQ}(\tau_i \mapsto \tau'_i)$ in G.

Proof. Assume E is a white node without losing generality. By construction, $E \in E'[E_1/E_2]$ for some elements E_1 and E_2.

For any pair of corresponding parameters τ_i and τ'_i, the interesting case is when $\tau_i \neq \tau'_i$. Assume $\mathcal{T}(E) = (E_0, s_1, s_2, \ldots, s_n)$, where s_i's are substitutions. Since component substitution does not change the top-level structure, the black node E_0 must have the form $\mathsf{con}\ \overline{\tau_0}$ (or $\mathsf{fun}\ \overline{\tau_0}$). By construction, τ_{0i} is a black node in G. Hence, the algorithm also adds τ_i by applying the same substitutions on τ_{0i}, as well as τ'_i by applying one more substitution $\tau_{0i}[E_1/E_2]$. $\mathsf{LEQ}(\tau_i \mapsto \tau'_i)$ is also added by saturation rules. □

Lemma 5 (Path simulation-Cons). *For any LEQ path from E_1 to E_2 in a fully saturated and expanded graph G where $E_1 = \mathsf{con}\ \overline{\tau_1}$ and $E_2 = \mathsf{con}\ \overline{\tau_2}$ for some constructor* con. *If at least one end node is white, then for any pair of corresponding parameters τ_i and τ'_i, either $\tau_i = \tau'_i$, or there is a path from τ_i to τ'_i in G.*

Proof. We prove by induction on the path length. The base case (length=1) is trivial by Lemma 4.

Assume the conclusion holds for any path whose length $\leq k$. Consider a path with length $k + 1$. Without losing generality, we assume E_1 is a white node.

Since a white node only connects elements with same top-level constructor, the path from E_1 to E_2 has the form: $\mathsf{con}\ \overline{\tau_1} - \mathsf{con}\ \overline{\tau_0} - \mathsf{con}\ \overline{\tau_2}$ for some τ_0. Result is true by Lemma 4 and induction hypothesis unless both $\mathsf{con}\ \overline{\tau_0}$ and $\mathsf{con}\ \overline{\tau_2}$ are black nodes.

When both $\mathsf{con}\ \overline{\tau_0}$ and $\mathsf{con}\ \overline{\tau_2}$ are black, all of their parameters are black by graph construction. Moreover, there is a path on each pair (τ_{0i}, τ_{i2}) by the second production in Figure 13. By Lemma 4, there is an edge connecting τ_{i1} and τ_{0i}. Therefore, there is a path from τ_{i1} to τ_{i2} if they are different. □

Lemma 6 (Path simulation-Funs). *For any LEQ path from E_1 to E_2 in a fully saturated and expanded graph G where $E_1 = \mathsf{fun}\ \overline{\tau_1}$ and $E_2 = \mathsf{fun}\ \overline{\tau_2}$ for some function* fun. *If for any edge along the path, at least one end node is white, then for any pair of corresponding parameters τ_i and τ'_i, either $\tau_i = \tau'_i$, or there is a path from τ_i to τ'_i in G.*

Proof. We prove by induction on the path length. The base case (length=1) is trivial by Lemma 4.

Assume the conclusion holds for any path whose length $\leq k$. Consider a path with length $k + 1$.

By assumption, every edge has at least one white node. Since a white node only connects elements with same top-level functions, the path from E_1 to E_2 has the form: $\mathsf{fun}\ \overline{\tau_1} - \mathsf{fun}\ \overline{\tau'} - \mathsf{fun}\ \overline{\tau_2}$ for some τ'. Result is true by Lemma 4 and induction hypothesis. □

Proof of Lemma 3. For any redundant path from E_1 to E_2, there exist non-redundant paths in G, say P_i from E_{i1} to E_{i2}, so that $E_{11} \leq E_{12} \wedge \ldots \wedge E_{n1} \leq E_{n2} \Leftrightarrow E_1 \leq E_2$.

Proof.

(1) When $E_1 = \mathsf{con}\ \overline{\tau_1}$ and $E_2 = \mathsf{con}\ \overline{\tau_2}$ for some constructor con.
We construct the desired set of non-redundant paths, say $\mathcal{P}$, as follows. For each parameter pair τ_{1i} and τ_{2i}, either $\tau_{1i} = \tau_{2i}$ or there is a path from τ_{1i} to τ_{2i} in G. We add nothing to $\mathcal{P}$ if $\tau_{1i} = \tau_{2i}$. Otherwise, we add the path to $\mathcal{P}$ if it is non-redundant. If the path is redundant, we recursively add all non-redundant paths that determines $\tau_{1i} \leq \tau_{2i}$ to $\mathcal{P}$. Easy to check $\mathcal{P}$ has the desired property and the recursion terminates since all elements are finite.

(2) When $E_1 = \mathsf{fun}\ \overline{\tau_1}$ and $E_2 = \mathsf{fun}\ \overline{\tau_2}$ for some constructor fun.
Similar to the case above, except we use Lemma 6 instead of Lemma 5 in the proof.

□

Since redundant edges provides neither positive nor negative evidence for error explanation, for brevity, we subsequently use the term path to mean a path that is non-redundant.

9 BAYESIAN MODEL FOR RANKING EXPLANATIONS

The observed symptom of errors is a fully analyzed constraint graph (Section 8), in which all informative LEQ edges are classified as satisfiable or unsatisfiable. For simplicity, in what follows we write "edge" to mean "informative and non-redundant edge".

Although the information along unsatisfiable paths already captures why a goal is unsatisfiable, reporting all constraints along a path may give more information than the programmer can digest. Our approach is to use Bayesian reasoning to identify programmer errors more precisely.

9.1 A Bayesian interpretation

The cause of errors can be wrong constraints, missing hypotheses, or both. To keep our diagnostic method as general as possible, we avoid building in domain-specific knowledge about mistakes programmers tend to make. However, the framework does permit adding such knowledge in a straightforward way.

The language-level entity about which errors are reported can be specific to the language. OCaml reports typing errors in expressions, whereas Jif reports errors in information-flow constraints. To make our diagnosis approach general, we treat entities as an abstract set Ω and assume a mapping Φ from entities to constraints. We assume a prior distribution on entities P_Ω, defining the probability that an entity is wrong. Similarly, we assume a prior distribution P_Ψ on hypotheses Ψ, defining the probability that a hypothesis is missing.

Given entities $E \subseteq \Omega$ and hypotheses $H \subseteq \Psi$, we are interested in the probability that E and H are the cause of the error observed. In this case, the observation o is the satisfiability of informative paths within the program. We denote the observation as $o = (o_1, o_2, \ldots, o_n)$, where $o_i \in \{\mathtt{unsat}, \mathtt{sat}\}$ represents unsatisfiability or satisfiability of the corresponding path. The observation follows some unknown distribution P_O.

We are interested in finding a subset E of entities Ω and a subset H of hypotheses Ψ for which the posterior probability $P(E, H|o)$ is large, meaning that E and H are likely causes of the given observation o. In particular, a *maximum a priori* estimate is a pair (E, H) at which the posterior probability takes its maximum value; that is, at $\arg\max_{E\subseteq\Omega, H\subseteq\Psi} P(E, H|o)$.

By Bayes' theorem, $P(E, H|o)$ is equal to

$$P_{\Omega\times\Psi}(E, H)P(o|E, H)/P_O(o)$$

The factor $P_O(o)$ does not vary in the variables E and H, so it can be ignored. Assuming the prior distributions on Ω and Ψ are independent, a simplified term can be used:

$$P_\Omega(E)P_\Psi(H)P(o|E, H)$$

$P_\Omega(E)$ is the prior knowledge of the probability that a set of entities E is wrong. In principle, this term might be estimated by learning from a large corpus of buggy programs or using language-specific heuristics. For simplicity and generality, we assume that each entity is equally likely to be wrong. We note that incorporating language-specific knowledge to refine the prior distribution $P_\Omega(E)$ (e.g., assigning weights to different kinds of constraints) will likely improve the accuracy of SHErrLoc, but we leave that to future work.

We also assume the probability of each entity being the cause is independent.[3] Hence, $P_\Omega(E)$ is estimated by $P_1^{|E|}$, where P_1 is a constant representing the likelihood that a single entity is wrong.

$P_\Psi(H)$ is the prior knowledge of the probability that hypotheses H are missing. Of course, not all hypotheses are equally likely to be wrong. For example, the hypothesis $\top \leq \bot$ is too strong to be useful: it makes all constraints succeed. The likely missing hypothesis is both weak and small. Our general heuristics for obtaining this term are discussed in Section 9.3.

$P(o|E,H)$ is the probability of observing the constraint graph, given that entities E are wrong and hypotheses H are missing. To estimate this factor, we assume that the satisfiability of the remaining paths is independent (again, refining this simplifying assumption will likely improve the accuracy of SHErrLoc). This allows us to write $P(o|E,H) = \prod_i P(o_i|E,H)$. The term $P(o_i|E,H)$ is calculated using two heuristics:

(1) For an unsatisfiable path, either something along the path is wrong, or adding H to the hypotheses on the path makes the partial ordering on end nodes valid. So when neither p_i has an entity in E, nor adding H to hypotheses on p_i makes it satisfiable, $P(o_i = \mathtt{unsat}|E,H) = 0$ and $P(o_i = \mathtt{sat}|E,H) = 1$.
(2) A satisfiable path is unlikely (with some constant probability $P_2 < 0.5$) to contain a wrong entity. Therefore, if path p_i contains a constraint generated by some entity in E, we have $P(o_i = \mathtt{sat}|E,H) = P_2$ and $P(o_i = \mathtt{unsat}|E,H) = 1 - P_2$.

The first heuristic suggests we only need to consider the entities and hypotheses that explain all unsatisfiable paths (otherwise $P(o_i|E,H) = 0$ for some $o_i = \mathtt{unsat}$ by heuristic 1). We denote this set by $\mathcal{G}$. Suppose entities E appear on N_E paths in total, among which k_E paths are satisfiable by definition. We say entities E cut a path p iff p contains some entity in E; hypotheses H explain a path p iff adding H to the hypotheses on p makes the partial ordering on end nodes valid. Then, based on the simplifying assumptions made, we have

$$\begin{aligned}
&\mathop{\arg\max}_{E\subseteq\Omega, H\subseteq\Psi} P_\Omega(E)P_\Psi(H)P(o|E,H)\\
=\ &\mathop{\arg\max}_{E\subseteq\Omega, H\subseteq\Psi} P_1^{|E|}P_\Psi(H)\Pi_i P(o_i|E,H)\\
=\ &\mathop{\arg\max}_{E\subseteq\Omega, H\subseteq\Psi} P_1^{|E|}P_\Psi(H)(\Pi_{i:E\text{ cut }p_i}P(o_i|E,H)\cdot\Pi_{i:\neg(E\text{ cut }p_i)\wedge\neg(H\text{ explain }p_i)}P(o_i|E,H)\\
&\qquad\qquad\cdot\Pi_{i:\neg(E\text{ cut }p_i)\wedge(H\text{ explain }p_i)}P(o_i|E,H))\\
=\ &\mathop{\arg\max}_{E\subseteq\Omega, H\subseteq\Psi} P_1^{|E|}P_\Psi(H)P_2^{k_E}(1-P_2)^{N_E-k_E}\Pi_{i:\neg(E\text{ cut }p_i)\wedge\neg(H\text{ explain }p_i)}P(o_i|E,H)\\
=\ &\mathop{\arg\max}_{(E,H)\in\mathcal{G}} P_1^{|E|}(P_2/(1-P_2))^{k_E}(1-P_2)^{N_E}P_\Psi(H)
\end{aligned}$$

[3]It seems likely that the precision of our approach could be improved by refining this assumption, since the (rare) missed locations in our evaluation usually occur when the programmer makes a similar error multiple times.

In the third equation, we drop the case $\neg(E \text{ cuts } p_i) \wedge (H \text{ explains } p_i)$ because H always explain a path that is already satisfiable. Therefore, H being missing hypotheses does not affect the prior distribution $P(o_i|E, H)$ in this case.

For simplicity, we approximate the most likely error causes by assuming N_E is roughly the same for all candidates in set $\mathcal{G}$. Hence, if $C_1 = -\log P_1$ and $C_2 = -\log(P_2/(1-P_2))$, maximizing the likelihood is equivalent to minimizing the ranking metric $|E| + (\frac{C_2}{C_1})k_E$. An intuitive understanding is that the cause must explain all unsatisfiable edges; the wrong entities are likely to be small ($|E|$ is small) and not used often on satisfiable edges (since $C_2 > 0$ by heuristic 2); the missing hypothesis is likely to be weak and small, as defined in Section 9.3, which maximizes the term $P_\Psi(H)$.

Notice that other than $|E|$ and k_E, the ranking metric only depends on the ratio between C_2 and C_1. Empirical results show that the ranking of expression sets according to this metric is insensitive to the value of C_2/C_1 for both OCaml (Section 10.2.3) and Haskell (Section 10.3.3) programs. This result suggests that SHErrLoc is likely to provide precise error locations for various applications without language-specific tunings.

9.2 Inferring likely wrong entities

The term $P_1^{|E|}(P_2/(1-P_2))^{k_E}$ can be used to calculate the likelihood that a subset of entities is the cause. However, its computation for all possible sets of entities can be impractical. Therefore, we propose an instance of A* search [22], based on novel heuristics, to calculate optimal solutions in a practical way.

A* search is a heuristic search algorithm for finding minimum-cost solution nodes in a graph of search nodes. In our context, each search node n represents a set of entities deemed wrong, denoted E_n. A solution node is one that explains all unsatisfiable paths—the corresponding entities appear in all unsatisfiable paths. An edge corresponds to adding a new entity to the current set.

The key to making A* search effective is a good cost function $f(n)$. The cost function is the sum of two terms: $g(n)$, the cost to reach node n, and $h(n)$, a *heuristic function* estimating the cost from n to a solution.

Before defining the cost function $f(n)$, we note that maximizing the likelihood $P_1^{|E|}(P_2/(1-P_2))^{k_E}$ is equivalent to minimizing $C_1|E| + C_2 k_E$, where $C_1 = -\log P_1$ and $C_2 = -\log(P_2/(1-P_2))$ are both positive constants because $0 < P_1 < 1$ and $0 < P_2 < 0.5$. Hence, the cost of reaching n is

$$g(n) = C_1|E_n| + C_2 k_{E_n}$$

To obtain a good estimate of the remaining cost—that is, the heuristic function $h(n)$—our insight is to use the number of entities required to cover the remaining unsatisfiable paths, denoted as $\mathcal{P}_{\mathtt{rm}}$, since C_1 is usually larger than C_2. More specifically, $h(n) = 0$ if $\mathcal{P}_{\mathtt{rm}} = \emptyset$. Otherwise, $h(n) = C_1$ if $\mathcal{P}_{\mathtt{rm}}$ is covered by one single entity; $h(n) = 2C_1$ otherwise.

An important property of the heuristic function is its optimality: all and only the most likely wrong subsets of entities are returned. This result is proven in Lemma 7. The heuristic search algorithm is also efficient in practice: on current hardware, it takes about 10 seconds even when the size of the search space is 2^{1000}. More performance details are given in Section 10.

Lemma 7. *The heuristic search algorithm is optimal.*

Proof. Since the heuristic search algorithm is based on A* search, we only need to show $h(n)$ is *consistent*. That is, for every node n and every successor n' of n, we have $h(n) \leq c(n, n') + h(n')$, where $c(n, n')$ is the cost from n to n'.

Since the successor always contains one more program entity, we have $C_1 \leq c(n, n')$. When $0 \leq h(n) \leq C_1$, we have $h(n) \leq C_1 + 0 \leq c(n, n) + h(n')$ for all n, n'. When $h(n) = 2C_1$, we know that

$h(n') \geq C_1$ for any successor n' of n by the design of $h(n)$. Hence, $h(n) = C_1 + C_1 \leq c(n, n') + h(n')$ when $h(n) = 2C_1$. □

Algorithm. The algorithm maintains a priority queue Q, a set of solutions S, and the minimum solution cost *min*. To avoid duplicated search states, we assume without loss of generality that entities in Ω are associated with unique identifiers. The algorithm works as follows. Notice that the algorithm returns all optimal explanations.

(1) Initially, Q contains a single node n_0 representing the empty set, $S = \emptyset$, and *min* is set to infinity.
(2) At each step, the algorithm removes a node n with the smallest cost from Q, w.r.t. the cost function $f(n) = g(n) + h(n)$ that we have define above, and tests whether E_n covers all unsatisfiable paths.
   (a) If E_n is a cover, add E_n to S if $f(n) \leq min$, and set *min* to $f(n)$ when *min* is infinity. If $f(n) > min$, goto step 3.
   (b) Otherwise, for each entity $e \in \Omega$ with an ID larger than any element in E_n, create a node n' where $E_{n'} = E_n \cup \{e'\}$, and add n' to Q. Then repeat step 2.
(3) Return set S.

9.3 Inferring missing hypotheses

Another factor in the Bayesian interpretation is the likelihood that hypotheses (assumptions) are missing. Recall that a path from element E_1 to E_2 in a constraint graph is unsatisfiable if the conjunction of hypotheses along the path is insufficient to prove the partial ordering $E_1 \leq E_2$. So we are interested in inferring a set of missing hypotheses that are sufficient to repair unsatisfiable paths in a constraint graph.

9.3.1 Motivating example. Consider the following assertions:

$$\begin{aligned} &(\texttt{Bob} \leq \texttt{Carol} \vdash \texttt{Alice} \leq \texttt{Bob}) \\ \wedge &(\texttt{Bob} \leq \texttt{Carol} \vdash \texttt{Alice} \leq \texttt{Carol}) \\ \wedge &(\texttt{Bob} \leq \texttt{Carol} \vdash \texttt{Alice} \leq \texttt{Carol} \sqcup \bot) \end{aligned}$$

Since the only hypothesis we have is $\texttt{Bob} \leq \texttt{Carol}$ (meaning Carol is more privileged than Bob), none of the three constraints in the conclusion holds. One trivial solution is to add all invalid conclusions to the hypothesis. This approach would add $\texttt{Alice} \leq \texttt{Bob} \wedge \texttt{Alice} \leq \texttt{Carol} \wedge \texttt{Alice} \leq \texttt{Carol} \sqcup \bot$ to the hypotheses. However, this naive approach is undesirable for two reasons:

(1) An invalid hypothesis may invalidate the program analysis. For instance, adding an insecure information flow to the hypotheses can violate security. The programmer has the time-consuming, error-prone task of checking the correctness of every hypothesis.
(2) A program analysis may combine static and dynamic approaches. For instance, although most Jif label checking is static, some hypotheses are checked dynamically. So a large hypothesis may also hurt run-time performance.

It may also be tempting to select the minimal missing hypothesis, but this approach does not work well either: a single assumption $\top \leq \bot$ is always a minimal missing hypothesis for all unsatisfiable paths. Given $\top \leq \bot$, any partial order $E_1 \leq E_2$ can be proved since $E_1 \leq \top \leq \bot \leq E_2$. However, this assumption is obviously too strong to be useful.

Intuitively, we are interested in a solution that is both *weakest* and *minimal*. In the example above, our tool returns a hypothesis with only one constraint $\texttt{Alice} \leq \texttt{Bob}$: both weakest and minimal.

We now formalize the *minimal weakest missing hypothesis*, and give an algorithm for finding this missing hypothesis.

9.3.2 Missing hypothesis. Consider an unsatisfiable path P that supports an $(^{+}\mathsf{LEQ})$ edge $e = (^{+}\mathsf{LEQ})\{H\}(n_1 \mapsto n_2)$. For simplicity, we denote the hypothesis of P as $\mathcal{H}(P) = H$, and the conclusion $C(P) = n_1 \le n_2$.

We define a *missing hypothesis* as follows:

Definition 2. *Given unsatisfiable paths $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, a set of inequalities S is a missing hypothesis for $\mathcal{P}$ iff $\forall P_i \in \mathcal{P} \;.\; \mathcal{H}(P_i) \wedge \bigwedge_{I \in S} I \vdash C(P_i)$.*

Intuitively, adding all inequalities in the missing hypothesis to the assertions' hypotheses removes all unsatisfiable paths in the constraint graph.[4]

Example. Returning to the example in Section 9.3.1, it is easy to verify that `Alice` $\le$ `Bob` is a missing hypothesis that makes all of the assertions valid.

9.3.3 Finding a minimal weakest hypothesis. We are not interested in all missing hypotheses; instead, we want to find one that is both *minimal* and as *weak* as possible.

To simplify the notation, we further define the conclusion set of unsatisfiable paths $\mathcal{P}$ as the union of all conclusions: $C(\mathcal{P}) = \bigcup\{C(P_i) \mid P_i \in \mathcal{P}\}$.

The first insight is that the inferred missing hypothesis should not be too *strong*.

Definition 3. *For a set of unsatisfiable paths $\mathcal{P}$, a missing hypothesis S is no weaker than S' iff*

$$\forall I' \in S' \;.\; \exists P \in \mathcal{P} \;.\; \mathcal{H}(P) \wedge \bigwedge_{I \in S} I \vdash I'$$

That is, S is no weaker than S' if all inequalities in S' can be proved from S, using at most one existing hypothesis.

Given this definition, the first property we show is that every subset of $C(\mathcal{P})$ that forms a missing hypothesis is maximally weak:

Lemma 8. $\forall S \subseteq C(\mathcal{P})$. *$S$ is a missing hypothesis $\Rightarrow$ no missing hypothesis is strictly weaker than S.*

Proof. Suppose there exists a strictly weaker missing hypothesis S'. Since S' is a missing hypothesis, $\mathcal{H}(P_i) \wedge \bigwedge_{I' \in S'} I' \vdash C(P_i)$ for all i. Since $S \subseteq C(\mathcal{P})$, $\forall I \in S \;.\; \mathcal{H}(P_i) \wedge \bigwedge_{I' \in S'} I' \vdash I$. So S' is no weaker than S. Contradiction. □

The lemma above suggests that subsets of $C(\mathcal{P})$ may be good candidates for a weak missing hypothesis. However, they are not necessarily minimal. For instance, the entire set $C(\mathcal{P})$ is a maximally weak missing hypothesis.

To remove the redundancy in this weakest hypothesis, we observe that some of the conclusions are subsumed by others. To be more specific, we say a conclusion c_i *subsumes* another conclusion $c_j = C(P_j)$ if $c_i \wedge \mathcal{H}(P_j) \vdash c_j$. Intuitively, if c_i subsumes c_j, then adding c_i to the hypothesis of P_j makes P_j satisfiable.

Example. Return to the example in Section 9.3.1. The missing hypothesis `Alice` $\le$ `Bob` is both the weakest and minimal.

Based on Lemma 8 and the definition above, finding a minimal weakest missing hypothesis in $C(\mathcal{P})$ is equivalent to finding the minimum subset of $C(\mathcal{P})$ which subsumes all $c \in C(\mathcal{P})$. This gives us the following algorithm:

[4] A more general form of missing hypothesis might infer individual hypotheses for each path. But it is less feasible to do so.

Algorithm. Given a set of unsatisfiable paths $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$:

(1) Construct the set $C(\mathcal{P})$ from the unsatisfiable paths.
(2) For all c_i, c_j in $C(\mathcal{P})$, add c_j to set S_i, the set of conclusions subsumed by c_i, if c_i subsumes c_j.
(3) Find the minimum cover M of $C(\mathcal{P})$, where $\mathcal{S} = \{S_1, \ldots, S_n\}$ and $M \subseteq \mathcal{S}$.
(4) Return $\{c_i \mid S_i \in M\}$.

A brute force algorithm for finding the minimal weakest missing hypothesis may check all possible hypotheses. That is on the order of 2^{N^2} (the number of all subsets of $\leq$ orderings on elements) where N is the total number of elements used in the constraints. While the complexity of our algorithm is exponential in the number of unsatisfiable paths in the constraint graph, this number is usually small in practice. So the computation is still feasible.

10 EVALUATION

10.1 Implementation

We implemented our general error diagnostic tool SHErrLoc in Java. SHErrLoc reads in constraints following the syntax of Figure 6, and computes constraints most likely to have caused errors in the constraint system being analyzed. The implementation includes about 8,000 lines of source code, excluding comments and blank lines. The SHErrLoc tool is released as open source [50].

To evaluate our error diagnostic tool on real-world program analyses, we modified several compilers to generate constraints in our constraint language format: Jif, EasyOCaml [15], and GHC [35]. EasyOCaml is an extension of OCaml 3.10.2 that generates the labeled constraints defined in [19].

Generating constraints in our constraint language format involved only at most modest effort. For Haskell type inference, little effort was required. We modified the GHC compiler (version 7.8.2), which already generates and solves constraints during type inference, to emit unsimplified, unsolved constraints. The modification is minimal: only 50 LOC (lines of code) are added or modified. Constraints in GHC's format are then converted by a lightweight Perl script (about 400 LOC) into the syntax of our error diagnosis tool.

Changes to the Jif compiler include about 300 LOC above more than 45,000 LOC in the unmodified compiler. Changes to EasyOCaml include about 500 LOC above the 9,000 LOC of the EasyOCaml extension. Slightly more effort is required for EasyOCaml because that compiler did not track the locations of type variables; this functionality had to be added so constraints could be traced back to the corresponding source code.

10.2 Case study: OCaml error reporting

To evaluate the quality of our ranking algorithm on OCaml, we used a corpus of previously collected OCaml programs containing errors, collected by Lerner et al. [31]. The data were collected from a graduate-level programming-language course for part-time students with at least two years professional software development experience. The data came from 5 homework assignments and 10 students participating in the class. Each assignment requires students to write 100–200 lines of code.

From the data, we analyzed only type mismatch errors, which correspond to unsatisfiable constraints. Errors such as unbound values or too many arguments to a constructor are more easily localized and are not our focus.

We also exclude programs using features not supported by EasyOCaml and files where the user's fix is unclear. After excluding these files, 336 samples remain.

10.2.1 Evaluation setup. Analyzing a file and the quality of error report message manually can be inherently subjective. We made the following efforts to make our analysis less subjective:

(1) Instead of judging which error message is more useful, we judged whether the error locations the tools reported were correct.
(2) To locate the actual error in the program, we use the user's changes with larger timestamps as a reference. Files where the error location is unclear are excluded in our evaluation.

To ensure the tools return precisely the actual error, a returned location is judged as correct only when it is a subset of the actual error locations.

One subtlety of judging correctness is that multiple locations can be good suggestions, because of let-bindings. For instance, consider a simple OCaml program: `let x = true in x + 1`.

Even if the programmer later changed `true` to be some integer, the error suggestion of the let-binding of `x` and the use of `x` are still considered to be correct since they bind to the same expression as the fix. However, the operation `+` and the integer `1` are not since the fix is not related.

Since the OCaml error message reports an expression that appears to have the wrong type, to make the reports comparable, we use expressions as the program entities on which we run our inference algorithm—our tool reports likely wrong expressions in evaluation. Recall that our tool can also generate reports of why an expression has a wrong type, corresponding to unsatisfiable paths in the constraint graph. Using such extra information might improve the error message, but we do not use that capability in the evaluation.

Another mismatch is that our tool inherently reports a small set of program entities (expressions in this case) with the same estimated quality, whereas OCaml reports one error at one time. To make the comparison fair, we make the following efforts:

(1) For cases where we report a better result (our tools finds the error location that OCaml misses), we ensure that all locations returned are correct.
(2) For other cases, we ensure that the majority of the suggestions are correct.

Moreover, the average top rank suggestion size is smaller than 2. Therefore, our evaluation results should not be affected much by the fact that our tool can offer multiple suggestions.

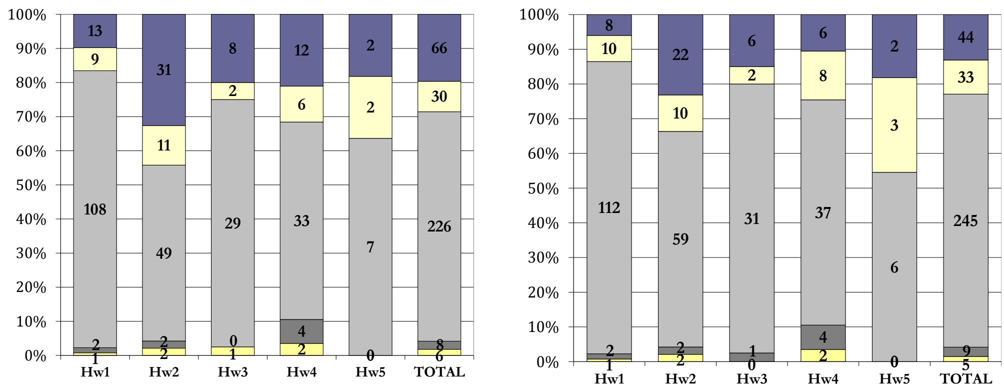
10.2.2 Error report accuracy. For each file we analyze, we consider both the error location reported by OCaml and the top-ranked suggestion of our tool (based on the setting $P_1 = (P_2/1-P_2)^3$). We reused the data offered by the authors of the Seminal tool [31], who labeled the correctness of Seminal's error location report.

We classify the files into one of the following five categories and summarize the results in Figure 19:

(1) Our approach suggests an error location that matches the programmer's fix, but the other tool's location misses the error.
(2) Our approach reports multiple correct error locations that match the programmer's fix, but the other tool only reports one of them.
(3) Both approaches find error locations corresponding to the programmer's fix.
(4) Both approaches miss the error locations corresponding to the programmer's fix.
(5) Our tool misses the error location but the other tool captures it.

For category (2), we note that SHErrLoc and Seminal can report multiple suggested error locations, while OCaml reports one error location. We report a file in category (2) if the programmer's fix consists of multiple locations, and only SHErrLoc correctly localizes multiple of them.

The result shows that OCaml's reports find about 75% of the error locations but miss the rest. Seminal's reports on error locations are slightly better, finding about 80% of the error locations.



(a) Comparison with the OCaml compiler

(b) Comparison with Seminal

Fig. 19. Results organized by homework assignment. From top to bottom, columns represent programs where (1) our tool finds a correct error location that the other tool misses. (2) both approaches report the correct error location, but our tool reports multiple (correct) error locations; (3) both approaches report the correct error location; (4) both approaches miss the error location; (5) our tool misses the error location while the other tool identifies one of them. For every assignment, our tool does the best job of locating the error.

Compared with both OCaml and Seminal, our tool consistently identifies a higher percentage of error locations across all homeworks, with an average of 96% (categories (1), (2) and (3)).

In about 10% of cases, our tool identifies multiple errors in programs. According to the data, the programmers usually fixed these errors one by one since the OCaml compiler only reports one at a time. Reporting multiple errors at once may be more helpful.

10.2.3 Sensitivity. Recall that maximizing the likelihood of entities E being an error is equivalent to minimizing the term $C_1|E| + C_2 k_E$, where $C_1 = -\log P_1$ and $C_2 = -\log(P_2/(1 - P_2))$ (see, Section 9.2). Hence, the ranking is only affected by the ratio between C_1 and C_2.

To test how sensitive our tool is to the choice of C_1/C_2, we collect two important statistics for a wide range of the ratio values:

(1) the accuracy of SHErrLoc (number of programs where the actual error is found in top-rank suggestions/336 programs),
(2) the average number of suggestions in the top rank.

The result is summarized in Table 1.

We arrange the columns in Table 1 such that the ratios between C_1 and C_2 increases linearly. That is, for any $0 < P_2 < 0.5$, P_1 decreases exponentially from left to right. The last column corresponds to the special case when $C_2 = 0$ (i.e., $P_2 = 0.5$).

Empirically, the overall suggestion quality is best when $C_1/C_2 = 3$. However, the quality of the suggestions is close for any C_1 and C_2 s.t. $2 \leq C_1/C_2 \leq 6$; the results are not very sensitive to the choice of these parameters.

If satisfiable paths are ignored ($C_2 = 0$, that is, $P_2 = 0.5$), the number of suggestions in the top rank is much larger, and more errors are missing. Hence, using satisfiable paths is important to suggestion quality. Intuitively, feedback from satisfiable paths helps to prioritize suggestions with the same size (i.e., suggestions containing the same number of program expressions). Moreover,

	$\frac{C_1}{C_2} = 1$	$\frac{C_1}{C_2} = 2$	$\frac{C_1}{C_2} = 3$	$\frac{C_1}{C_2} = 4$	$\frac{C_1}{C_2} = 5$	$\frac{C_1}{C_2} = 6$	$\frac{C_1}{C_2} = 10$	$C_2 = 0$
Accuracy	94%	95%	96%	95%	95%	95%	94%	93%
Avg. # Sugg.	1.86	1.80	1.72	1.69	1.70	1.69	1.67	5.58

Table 1. The quality of top-ranked suggestions with various ratios between C_1 and C_2.

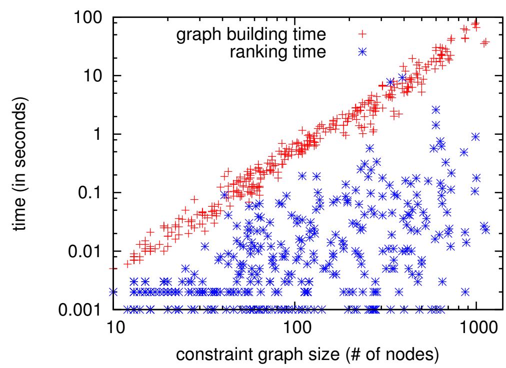


Fig. 20. Performance on the OCaml benchmark.

considering satisfiable paths improves accuracy for programs with multiple errors, since the top-ranked suggestions may not have the minimal size (due to the C_2k_E component).

The quality of the error report is also considerably worse when $C_1/C_2 = 1$. This result shows that unsuccessful paths are more important than successful paths, but that ascribing too importance to the unsuccessful paths (e.g., at $C_1/C_2 = 10$) also hurts the quality of the error report.

Limitations. Of course, our tool sometimes misses errors. We studied programs where our tool missed the error location, finding that in each case it involved multiple interacting errors. In some cases the programmer made a similar error multiple times. For those programs, our tool reports misleading locations (usually, one single location) that are not relevant to any of the error locations. One possible reason is that those programs violate the assumption of error independence. As our result suggests, this situation is rare.

The comparison between the tools is not completely apples-to-apples. We only collect type mismatch errors in the evaluation. OCaml is very effective at finding other kinds of errors such as unbound variables or wrong numbers of arguments, and Seminal not only finds errors but also proposes fixes.

10.2.4 Performance. We measured the performance of our tool on a Ubuntu 11.04 system using a dual core at 2.93GHz with 4G memory. Results are shown in Figure 20. We separate the time spent generating and inferring LEQ edges in the graph from that spent computing rankings.

The results show how the running time of both graph building time and ranking time scale with increasing constraint graph size. Interestingly, graph building, including the inference of ($^+$LEQ) relationships, dominates and is in practice quadratic in the graph size. The graph size has less impact on the running time of our ranking algorithm. We suspect the reason is that the running time of our ranking algorithm is dominated by the number of unsatisfiable paths, which is not strongly related to total graph size.

Considering graph construction time, all programs finish in 79 seconds, and over 95% are done within 20 seconds. Ranking is more efficient: all programs finish in 10 seconds. Considering the human cost to identify error locations, the performance seems acceptable.

10.3 Case study: Haskell type inference

To evaluate the quality of our ranking algorithm on Haskell, we used two sets of previously collected Haskell programs containing errors. The first corpus (the CE benchmark [10]) contains 121 Haskell programs, collected from 22 publications about type-error diagnosis. Although many of these programs are small, most of them have been carefully chosen or designed in the 22 publications to illustrate important (and often, challenging) problems for error diagnosis.

The second benchmark, the Helium benchmark [20], contains over 50,000 Haskell programs logged by Helium [23], a compiler for a substantial subset of Haskell, from first-year undergraduate students working on assignments of a programming course offered at the University of Utrecht during course years 2002–2003 and 2003–2004. Among these programs, 16,632 contain type errors.

10.3.1 Evaluation setup. To evaluate the quality of an error report, we first need to retrieve the *true error locations* of the Haskell programs being analyzed, before running our evaluation.

The CE benchmark contains 86 programs where the true error locations are well-marked. We reused these locations in evaluation. Since not all collected programs are initially written in Haskell, the richer type system of Haskell actually makes 9 of these programs type-safe. Excluding these well-typed programs, 77 programs are left.

The Helium benchmark contains programs written by 262 groups of students taking the course. To make our evaluation objective, we only considered programs whose true error locations are clear from subsequences of those programs where the errors are fixed. Among those candidates, we picked one program with the latest time stamp (usually the most complex program) for each group to make our evaluation feasible. Groups were ignored if either they contain no type errors, or the error causes are unclear. In the end, we used 228 programs. The programs were chosen without reference to how well various tools diagnosed their errors.

We compared the *error localization accuracy* of our tool to GHC 7.8.2 and Helium [25]; both represent the state of the art for diagnosing Haskell errors. A tool accurately locates the errors in a program if and only if it points to at least one of the true error locations in the program.

One difference from GHC and Helium is that sometimes, our tool reports a small set of top-rank source locations, with the same likelihood. For fairness, we ensure that the majority of suggestions are correct when we count our tool as accurate. Average suggestion size is 1.7, so we expect a limited effect on results for offering multiple suggestions.

10.3.2 Error report accuracy. Figure 21 shows the error report accuracy of our tool, compared with GHC and Helium. For the CE benchmark, our tool provides strictly more accurate error reports for 43% and 26% of the programs, compared with GHC and Helium respectively. Overall, GHC, Helium and our tool find the true error locations for 48%, 68% and 88% of programs respectively. Clearly, our tool, with no Haskell-specific heuristics, already significantly improves accuracy compared with tools that do.

On the Helium benchmark, the accuracy of GHC, 68%, is considerably better than on the CE benchmark; our guess is the latter offers more challenging cases for error diagnosis. Nevertheless, our tool still outperforms GHC by 21%. Compared with Helium, our tool is strictly better for 21% of the programs. Overall, the accuracy of our tool is 89% for the Helium benchmark, a considerable improvement compared with both GHC (68%) and Helium (75%).

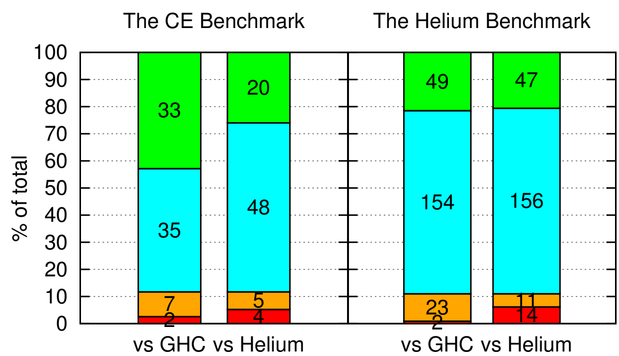


Fig. 21. Comparison with GHC and Helium on two benchmarks. From top to bottom, columns count programs where (1) our tool finds a correct error location that the other tool misses; (2) both tools report the correct error location; (3) both approaches miss the error location; (4) our tool misses the error location but the other tool finds one of them.

(a) CE benchmark

	$\frac{C_1}{C_2} = 1$	$\frac{C_1}{C_2} = 2$	$\frac{C_1}{C_2} = 3$	$\frac{C_1}{C_2} = 4$	$\frac{C_1}{C_2} = 5$	$C_2 = 0$
Accuracy	88%	88%	88%	88%	88%	90%
Avg. Sugg. Size	1.62	1.65	1.62	1.62	1.62	2.96

(b) Helium benchmark

	$\frac{C_1}{C_2} = 1$	$\frac{C_1}{C_2} = 2$	$\frac{C_1}{C_2} = 3$	$\frac{C_1}{C_2} = 4$	$\frac{C_1}{C_2} = 5$	$C_2 = 0$
Accuracy	89%	89%	89%	89%	88%	87%
Avg. Sugg. Size	1.71	1.73	1.73	1.73	1.68	2.52

Table 2. The quality of top-ranked suggestions with various ratios between C_1 and C_2.

Limitations. Our tool sometimes does miss error causes identified by other tools. For 14 programs, Helium finds true error locations that our tool misses. Among these programs, most (12) contain the same mistake: students confuse the list operators for concatenation (++) and cons (:). To find these error causes, Helium uses a heuristic based on the knowledge that this particular mistake is common in Haskell. It is likely that our tool, which currently uses no Haskell-specific heuristics, can improve accuracy further by exploiting knowledge regarding common mistakes. However, we leave integration of language-specific heuristics to future work.

Comparison with CF-typing. [10] evaluated their CF-typing method on the CE benchmark. For the 86 programs where the true error locations are well-marked, the accuracy of their tool is 67%, 80%, 88% and 92% respectively, when their tool reports 1, 2, 3 and 4 suggestions for each program; the accuracy of our tool is 88% with an average of 1.62 suggestions[5]. When our tool reports suboptimal suggestions, the accuracy becomes 92% , with an average suggestion size of 3.2.

[5] A slight difference is that we excluded 9 programs that are well-typed in Haskell. However, we confirmed that the accuracy of CF-typing on the same 77 programs changes by 1% at most [8].

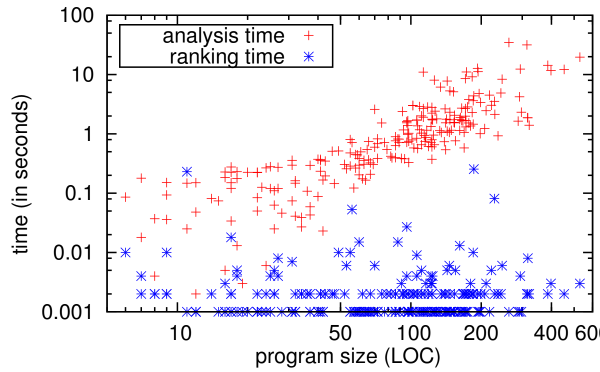


Fig. 22. Performance on the Helium benchmark.

10.3.3 Sensitivity. Recall (Section 9) that the only tunable parameter that affects ranking of error diagnoses is the ratio between C_2 and C_1. To see how the ratio affects accuracy, we measured the accuracy of our tool with different ratios (from 0.2 to 5), as summarized in Table 2. The result is that accuracy and average suggestion size of our tool change by at most 1% and 0.05 respectively. Hence, the accuracy of our tool does not depend on choosing the ratio carefully.

If only unsatisfiable paths are used for error diagnosis (i.e., $C_2 = 0$), the top-rank suggestion size is much larger (over 2.5 for both benchmarks, compared with ~1.7). Hence, satisfiable paths *are* important for error diagnosis.

10.3.4 Performance. We evaluated the performance of our tool on a Ubuntu 14.04 system with a dual-core 2.93GHz Intel E7500 processor and 4GB memory. We separate the time spent into that taken by graph-based constraint analysis (Section 6) and by ranking (Section 9).

The CE benchmark. Most programs in this benchmark are small. The maximum constraint analysis and ranking time for a single program are 0.24 and 0.02 seconds respectively.

The Helium benchmark. Figure 22 shows the performance on the Helium benchmark. The results suggest that both constraint analysis and ranking scale reasonably with increasing size of Haskell program being analyzed. Constraint analysis dominates the running time of our tool. Although the analysis time varies for programs of the same size, in practice it is roughly quadratic in the size of the source program.

Constraint analysis finishes within 35 seconds for all programs; 96% are done within 10 seconds, and the median time is 3.3 seconds. Most (on average, 97%) of the time required is used by graph saturation rather than expansion. Ranking is more efficient: all programs take less than one second.

We note that the results only apply to Haskell programs up to 600 LOC; the scalability of SHErrLoc might be a concern for larger programs. We leave optimizing SHErrLoc as future work.

10.4 Case study: Jif hypothesis inference

We also evaluated how helpful our hypothesis inference algorithm is for Jif. In our experience with using Jif, we have found missing hypotheses to be a common source of errors.

A corpus of buggy programs was harder to find for Jif than for OCaml and Haskell. We obtained application code developed for other, earlier projects using either Jif or Fabric (a Jif extension). These applications are interesting since they deal with real-world security concerns.

	Secure	Tie	Better	Worse	Total
Number	12	17	11	0	40
Percentage	30%	42.5%	27.5%	0%	100%

Table 3. Hypothesis inference result.

To mimic potential errors programmer would meet while writing the application, we randomly removed hypotheses from these programs, generating, in total, 40 files missing 1–5 hypotheses each. The frequency of occurrence of each application in these 40 files corresponds roughly to the size of the application.

For all files generated in this way, we classified each file into one of four categories, with the results summarized in Table 3:

(1) The program passed Jif/Fabric label checking after removing the hypotheses: the programmer made unneeded assumptions.
(2) The generated missing hypotheses matched the one we removed.
(3) The generated missing hypotheses provides an assumption that removes the error, but that is weaker than the one we removed (in other words, an improvement).
(4) Our tool fails to find a suggestion better than the one removed.

The number of redundant assumptions in these applications is considerable (30%). We suspect the reason is that the security models in these applications are nontrivial, so programmers have difficulty formulating their security assumptions. This observation suggests that the ability to automatically infer missing hypotheses could be very useful to programmers.

All the automatically inferred hypotheses had at least the same quality as manually written ones. This preliminary result suggests that our hypothesis inference algorithm is very effective and should be useful to programmers.

10.5 Case study: combined errors

To see how useful our diagnostic tool is for Jif errors that occur in practice, we used a corpus of buggy Fabric programs that a developer collected earlier during the development of the "FriendMap" application [4]. As errors were reported by the compiler, the programmer also clearly marked the nature and true location of the error. This application is interesting for our evaluation purposes since it is complex—it was developed over the course of six weeks by two developers—and it contains both types of errors: missing hypotheses and wrong expressions.

The corpus contains 24 buggy Fabric programs. One difficulty in working on these programs directly was that 9 files contained many errors. This happened because the buggy code was commented out earlier by the programmer to better localize the errors reported by the Fabric compiler. We posit that this can be avoided if a better error diagnostic tool, like ours, is used. For these files, we reproduced the errors the programmer pointed out in the notes when possible and ignored the rest. Redundancy—programs producing the same errors—was also removed. Result for the remaining 16 programs are shown in Table 4.

Most files contain multiple errors. We used the errors recorded in the note as actual errors, and an error is counted as being identified only when the actual error is suggested among top rank suggestions.

The first approach (Separate) measures errors identified if the error type is known ahead, or both hypothesis and expression suggestions separately computed are used. The result is comparable to the result in Sections 10.2 and 10.4, where error types are known ahead.

	Errors	Separate	Combined	Interactive
Missing hypothesis	11	10	7	11
Wrong expression	5	4	4	4
Total	16	14	11	15
Percentage	100%	87.5%	68.75%	93.75%

Table 4. Jif case study result. (1) Separate: top rank of both separately computed hypothesis and expression suggestions (2) Combined: top rank combined result only (3) Interactive approach.

Providing a concise and correct error report when multiple errors interact can be more challenging. We evaluated the performance of two approaches providing combined suggestions. The combined approach simply ranks the combined suggestions by size. Despite its simplicity, the result is still useful since this approach is automatic.

The interactive approach calculates missing hypotheses and requires a programmer to mark the correctness of these hypotheses. Then, correct hypotheses are used and wrong entities are suggested to explain the remaining errors. We think this approach is the most promising, since it involves limited manual effort: hypotheses are usually facts of properties to be checked, such as "is a flow from `Alice` to `Bob` secure?". We leave a more comprehensive study of this approach to future work.

11 RELATED WORK

Program analyses, constraints and graph representations. Modeling program analyses via constraint solving is not a new idea. The most related work is on set constraint-based program analysis [1, 2] and type qualifiers [17]. However, these constraint languages do not model hypotheses, which are important for some program analyses, such as information flow.

Program slicing, shape analysis, and flow-insensitive points-to analysis are expressible using graph-reachability [49]. Melski and Reps [38] show the interchangeability between context-free-language reachability (CFL-reachability) and a subset of set constraints [1]. But only a small set of constraints—in fact, a single variable—may appear on the right hand side of a partial order. Moreover, no error diagnostic approach is proposed for the graphs.

Error diagnoses for type inference and information-flow control. Dissatisfaction with error reports has led to earlier work on improving the error messages of both ML-like languages and Jif.

Efforts on improving type-error messages in ML-like languages can be traced to the early work of Wand [55] and of Johnson and Walz [27]. These two pieces of work represent two directions in improving error messages: the former traces *everything* that contributes to the error, whereas the latter attempts to infer the *most likely* cause. We only discuss the most related among them, but Heeren's summary [24] provides more details.

In the first direction, several efforts [11, 17, 19, 48, 52] improve the basic idea of Wand [55] in various ways. Despite the attractiveness of feeding a full explanation to the programmer, the reports are usually verbose and hard to follow [24].

In the second direction, one approach is to alter the order of type unification [9, 30, 36]. But since the error location may be used anywhere during the unification procedure, any specific order fails in some circumstance. Some prior work [21, 24, 27, 34, 45] also builds on constraints, but these constraint languages at most have limited support for sophisticated features such as type classes, type signatures, type families, and GADTs. Most of these approaches also use language-specific heuristics to improve report quality. For example, the accuracy of MinErrLoc [45] depends on the application-specific weight assigned to each constraint, while the accuracy of Mycroft [34] depends on the identification of certain hard constraints (i.e., constraints that should never be blamed) for OCaml type inference. As reported by Loncaric et al. [34], SHErrLoc achieves at least comparable accuracy on OCaml programs while treating all constraints as equally likely to be wrong.

A third approach is to generate fixes for errors by searching for similar programs [31, 37] or type substitutions [10] that do type-check. Unfortunately, we cannot obtain a common corpus to perform direct comparison with McAdam [37]. We are able to compare directly with the work of [31]; the results of Section 10.2 suggest that SHErrLoc improves on accuracy of [31] by 10%. Moreover, the results on the CE benchmark (Section 10.3.2) suggest that our tool localizes true error locations more accurately than the prior approach of Chen and Erwig [10]. Although SHErrLoc currently does not provide suggested fixes, accurate error localization is likely to provide good places to search for fixes. Combining these two techniques may be a fruitful area for future work.

For information-flow control, King et al. [28] propose to generate a trace explaining the information-flow violation. Although this approach also constructs a diagnosis from a dependency graph, only a subset of the DLM model is handled. As in type-error slicing, reporting whole paths can yield very verbose error reports. Recent work by Weijers et al. [57] diagnoses information-flow violations in a higher-order, polymorphic language. But the mechanism is based on tailored heuristics and a more limited constraint language. Moreover, the algorithm in [57] diagnoses a single unsatisfiable path, while our algorithm diagnoses multiple errors.

Probabilistic inference. Applying probabilistic inference to program analysis has appeared in earlier work, particularly on specification inference [29, 33]. Our contribution is to apply probabilistic inference to a general class of static analyses, allowing errors to be localized without language-specific tuning. Also related is work on statistical methods for diagnosing dynamic errors (e.g., [32, 60]). These algorithms rely on a different principle—statistical interpretation—and do not handle important features for static analysis, such as constructors and hypotheses.

The work of Ball et al. [5] on diagnosing errors detected by model checking has exploited a similar insight by using information about traces for both correct execution and for errors to localize error causes. Beyond differences in context, that work differs in not actually using probabilistic inference; each error trace is considered in isolation, and transitions are not flagged as causes if they lie on *any* correct trace.

Missing hypothesis inference. The most related work on inferring likely missing hypotheses is the recent work by Dillig et al. [14] on error diagnosis using abductive inference. This work computes small, relevant queries presented to a user that capture exactly the information a program analysis is missing to either discharge or validate the error. It does not attempt to identify incorrect constraints.

With regard to hypothesis inference, the Dillig algorithm infers missing hypotheses for a single assertion, whereas our tool finds missing hypotheses that satisfy a *set* of assertions. Further, the Dillig algorithm infers additional invariants on *variables* (e.g., $x \leq 3$ for a constraint variable x), while our algorithm also infers missing partial orderings on *constructors* (such as `Alice` $\leq$ `Bob` in Section 9.3.1).

Recent work by Blackshear and Lahiri [7] assigns confidence to errors reported by modular assertion checkers. This is done by the computation of an *almost-correct specification* that is used to identify errors likely to be false positives. This idea is largely complementary to our approach: although their algorithm returns a set of high-confidence errors, it does not attempt to infer their likely cause. At least for some program analyses, the heuristics they develop might also be useful for classifying whether errors result from missing hypotheses or from wrong constraints. As with the comparison to Dillig et al. [14], our algorithm also infers missing partial orderings on constructors, not just additional specifications on variables.

12 CONCLUSION

Better tools for helping programmers locate the errors detected by type systems and other program analyses should help adoption of the many powerful program analyses that have been developed. The science of diagnosing programmer errors is still in its infancy, but this article takes a step towards improving the situation. Our analysis of program constraint graphs offers a general, principled way to identify both incorrect expressions and missing assumptions. Results on three very different languages, OCaml, Haskell and Jif, with no language-specific customization, suggest this approach is promising and broadly applicable.

There are many interesting directions to take this work. Though we have shown that the technique works well on three very different type systems, it would likely be fruitful to apply these ideas to other type systems and program analyses, such as dataflow analysis and points-to analysis as we sketched in this article, and to explore more sophisticated ways to estimate the likelihood of different error explanations by incorporating prior knowledge about likely errors.

REFERENCES

[1] Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35 (1999), 79–111.

[2] Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Conf. Functional Programming Languages and Computer Architecture*. 31–41.

[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. DIKU, University of Copenhagen.

[4] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. 2012. Sharing Mobile Code Securely With Information Flow Control. In *IEEE Symp. on Security and Privacy*. 191–205.

[5] Thomas Ball, Mayur Naik, and Sriram Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *ACM Symp. on Principles of Programming Languages (POPL)*. 97–105.

[6] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30 (2000), 809–837.

[7] Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-correct Specifications: A Modular Semantic Framework for Assigning Confidence to Warnings. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 209–218.

[8] Shen Chen. 2014. Accuracy of CF-Typing. Private communication. (2014).

[9] Sheng Chen and Martin Erwig. 2014a. *Better Type-Error Messages Through Lazy Typing.* Technical Report. Oregon State University.

[10] Sheng Chen and Martin Erwig. 2014b. Counter-Factual Typing for Debugging Type Errors. In *ACM Symp. on Principles of Programming Languages (POPL)*.

[11] Venkatesh Choppella and Christopher T. Haynes. 1995. *Diagnosis of Ill-typed Programs.* Technical Report. Indiana University.

[12] Luis Manuel Martins Damas. 1985. *Type assignment in programming languages.* Ph.D. Dissertation. Department of Computer Science, University of Edinburgh.

[13] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. 19, 5 (1976), 236–243.

[14] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 181–192.

[15] EasyOCaml 2009. EasyOCaml. http://easyocaml.forge.ocamlcore.org. (2009).

[16] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. 1997. *Flow-Insensitive Points-to Analysis with Term and Set Constraints.* Technical Report. Berkeley, CA, USA.

[17] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. 2006. Flow-Insensitive Type Qualifiers. 28, 6 (Nov. 2006), 1035–1087.

[18] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. 2004. *Bayesian Data Analysis* (2nd ed.). Chapman & Hall/CRC.

[19] Christian Haack and J. B. Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming* 50, 1–3 (2004), 189–224.

[20] Jurriaan Hage. 2014. Helium benchmark programs, (2002–2005). Private communication. (2014).

[21] Jurriaan Hage and Bastiaan Heeren. 2007. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages*, Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.). Lecture Notes in Computer Science, Vol. 4449. Springer, 199–216.

[22] P.E. Hart, N.J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on* 4, 2 (1968), 100–107.

[23] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proc. 2003 ACM SIGPLAN Workshop on Haskell*. 62–71.

[24] Bastiaan J. Heeren. 2005. *Top Quality Type Error Messages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands.

[25] Helium 1.8(2014) 2014. Helium (ver. 1.8). https://hackage.haskell.org/package/helium. (2014).

[26] Paul Hudak, Simon Peyton Jones, and Philip Wadler. 1992. Report on the programming language Haskell. *SIGPLAN Notices* 27, 5 (May 1992).

[27] Gregory F. Johnson and Janet A. Walz. 1986. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages (POPL)*. 44–57.

[28] Dave King, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. 2008. Effective blame for information-flow violations. In *Int'l Symp. on Foundations of Software Engineering*. 250–260.

[29] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: inferring the specification within. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 161–176.

[30] Oukseh Lee and Kwangkeun Yi. 1998. Proofs About a Folklore Let-Polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 707–723.

[31] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 425–434.

[32] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 15–26.

[33] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 75–86.

[34] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. 781–799.

[35] Simon Marlow and Simon Peyton-Jones. 1993. The Glasgow Haskell Compiler. http://www.aosabook.org/en/ghc.html. (1993).

[36] Bruce James McAdam. 1998. On the Unification of Substitutions in Type Inference. In *Implementation of Functional Languages*. 139–154.

[37] Bruce James McAdam. 2001. *Repairing Type Errors in Functional Programs*. Ph.D. Dissertation. Laboratory for Foundations of Computer Science, The University of Edinburgh.

[38] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science* 248, 1–2 (2000), 29–98.

[39] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

[40] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *ACM Symp. on Operating System Principles (SOSP)*. 129–142.

[41] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java Information Flow. Software release, www.cs.cornell.edu/jif. (July 2006).

[42] Anil Nerode and Richard Shore. 1997. *Logic for Applications (2nd edition)*. Springer, New York, NY.

[43] OCaml 2016. OCaml programming language. http://ocaml.org. (2016).

[44] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theor. Pract. Object Syst.* 5, 1 (Jan. 1999), 35–55.

[45] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *2014 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 525–542.

[46] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82. DOI: http://dx.doi.org/10.1017/S0956796806006034

[47] Francois Pottier and Didier Rémy. 2005. The essence of ML type inference. In *Advanced topics in types and programming languages*, Benjamin C. Pierce (Ed.). MIT Press, 389–489.

[48] Vincent Rahli, J. B. Wells, and Fairouz Kamareddine. 2010. *A constraint system for a SML type error slicer*. Technical Report HW-MACS-TR-0079. Heriot-Watt university.

[49] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11–12 (1998), 701–726.

[50] SHErrLoc 2014. SHErrLoc (Static Holistic Error Locator) Tool Release (ver 1.0). http://www.cs.cornell.edu/projects/sherrloc. (2014).

[51] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *ACM Symp. on Principles of Programming Languages (POPL)*. 32–41.

[52] Frank Tip and T. B. Dinesh. 2001. A Slicing-Based Approach for Locating Type Errors. *ACM Trans. on Software Engineering and Methodology (TOSEM)* 10, 1 (2001), 5–55.

[53] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming* (September 2011).

[54] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Proc. 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, New York, NY, USA, 39–50. DOI: http://dx.doi.org/10.1145/1708016.1708023

[55] Mitchell Wand. 1986. Finding the Source of Type Errors. In *ACM Symp. on Principles of Programming Languages (POPL)*.

[56] Mitchell Wand. 1987. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae* 10 (1987), 115–122.

[57] Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. 2013. Security type error diagnosis for higher-order, polymorphic languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 3–12.

[58] Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. In *ACM Symp. on Principles of Programming Languages (POPL)*. 569–581.

[59] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. Diagnosing Type Errors with Class. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 12–21.

[60] Alice X. Zheng, Ben Liblit, and Mayur Naik. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *ICML'06*. 1105–1112.

Received August 2016; revised June 2017; accepted June 2017