Toward General Diagnosis of Static Errors

Danfeng Zhang   Andrew C. Myers
Department of Computer Science
Cornell University
Ithaca, NY, 14853
zhangdf@cs.cornell.edu   andru@cs.cornell.edu

Abstract

We introduce a general way to locate programmer mistakes that are detected by static analyses such as type checking. The program analysis is expressed in a constraint language in which mistakes 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 for two very different program analyses: type inference in OCaml and information flow checking in Jif. 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, the general technique identifies the location of programmer errors significantly more accurately.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Diagnostics; D.4.6 [*Security and Protection*]: Information flow controls; F.3.2 [*Semantics of Programming Languages*]: Program analysis.

Keywords Error diagnosis; static program analysis; type inference; information flow

1. Introduction

Sophisticated type systems and other program analyses enable verification of complex, important properties of software. Advances in type inference, dataflow analysis, and constraint solving have made these verification methods more practical by reducing both analysis time and annotation burden. However, the impact on industrial practice is disappointing.

We posit that a key barrier to adoption of sophisticated analyses is that debugging is difficult when the analysis reports an error. 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. Determining from this 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 two programming languages: ML, whose unification-based type inference algorithm sometimes generates complex, even misleading error messages [36], and Jif [31], a version of Java that statically analyzes the security of information flow within programs but whose error messages also confuse programmers [20]. 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. 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.

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 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., [8, 15, 35, 36]). But such reports are often verbose and hard to understand [17].

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.

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.

Contributions This paper presents the following contributions:

1. A general constraint language that can express a broad range of program analyses. We show that it can encode both ML type inference and Jif information flow analysis, as well as other analyses, including many dataflow analyses (Section 3).
2. A general algorithm for identifying likely program errors, based on the analysis of a constraint system extracted from the pro-

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535870

```
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)])
```

Figure 1. OCaml example. Line 9 is blamed for a mistake at line 7.

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

Figure 2. Jif example. Line 3 is blamed for a mistake at line 1.

gram. Using a Bayesian posterior distribution [14], the algorithm suggests program expressions that are likely errors and offers hypotheses that the programmer is likely to have omitted (Sections 4 and 5).

3. An evaluation of this new error diagnosis algorithm on two different sets of programs written in OCaml and Jif. As part of this evaluation we use a large set of programs collected from students using OCaml to do programming assignments [23] (Section 6). Appealingly, high-quality results do not rely on language-specific tuning.

2. Approach

Our general approach to diagnosing errors can be illustrated through examples from two languages: ML 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 [23]. 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 [32], SML [29], and Haskell [18]) 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 [8, 15, 35, 36]. 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.

2.2 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 [31] is a Java-like language whose static analysis of information flow often generates confusing error messages [20]. Figure 2 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. 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 $\{\texttt{this}\}$ 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.

Jif permits programmers to 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).

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. That assumption would imply that an information flow from patient to doctor is secure.

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

2.3 Overview of the approach

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

Constraints in this language assert partial orderings on constraint elements. These 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, Figure 3 is part of the constraint graph generated from the OCaml code of Figure 1. Each node represents either the type of a program expression or a declared type; in the figure, nodes are annotated with the line numbers of that expression or declaration. Each solid edge represents one constraint generated by an OCaml typing rule. For example, the leftmost node represents the type of the result of `print_string`, which is `unit`. Since function `loop` can return this result, the leftmost node is connected by edges to the node representing the result type of `loop` (at line 9).

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

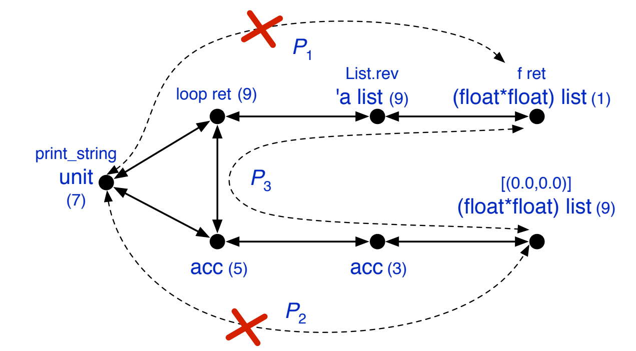


Figure 3. Part of the constraint graph for the OCaml example

represent the same types. Other such inferred paths exist, such as between the node for `unit` and the node for variable $acc_{(3)}$, 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.

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.

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 paths.

In Figure 3, 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. The `unit` node appears only on unsatisfiable informative paths, but not on the informative, satisfiable path P_3. Since erroneous nodes are less likely to appear in satisfiable paths, the `unit` node is a better error explanation than any node lying on path P_3.

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: our tool correctly and precisely points out the actual error in both the OCaml and Jif examples above.

In addition to helping identify incorrect expressions, the constraint graph also provides enough information to identify assumptions that are likely to be missing.

3. Constraint language

Central to our approach is a general core constraint language 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.

3.1 Syntax

The syntax of the constraint language is formalized in Figure 4.

$$G ::= G_1 \wedge G_2 \mid A \qquad A ::= C_1 \vdash C_2$$
$$C ::= I_1 \wedge \ldots \wedge I_n \;\; {}^{(n \geq 0)} \qquad I ::= E_1 \leq E_2$$
$$E ::= \alpha \mid c(E_1, \ldots, E_{a(c)}) \mid \overline{c}^i(E) \mid E_1 \sqcup E_2 \mid E_1 \sqcap E_2 \mid \bot \mid \top$$

Figure 4. Syntax of constraints

The top-level goal G to be solved is a conjunction of assertions A, each with the form $C_1 \vdash C_2$, where constraint C_1 is the hypothesis (that is, assumption) and constraint C_2 is a conclusion to be satisfied.

A constraint C, serving either as the hypothesis or as the conclusion of an assertion, is a possibly empty conjunction of inequalities I over elements from E, based on the ordering $\leq$. We denote an empty conjunction as $\emptyset$, and abbreviate $\emptyset \vdash C_2$ as $\vdash C_2$.

An element E may be a variable $\alpha \in \mathbf{Var}$ whose value is to be solved for, an application of constructor $c \in \mathbf{Con}$ or the i-th argument to a constructor application, represented by $\overline{c}^i(E)$. The arity of constructor c is represented as $a(c)$. Constants c are nullary constructors, with arity 0.

The ordering $\leq$ is treated abstractly, but it must define a lattice with the usual join ($\sqcup$) and meet ($\sqcap$) operators, which can be used as syntax. The bottom and top of the element ordering are $\bot$ and $\top$.

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. 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 3.4.1 describes in more detail how assertions are generated for ML.

3.2 Interpretation of constraints

The partial ordering on two applications of the same constructor is determined by the variances 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.

More general partial ordering rules on constructors (e.g., a rule $c_1(x, y) \leq c_2(y, x)$ can also be handled by our inference algorithm in 4.3 in a manner similar to the handling of $\sqcap$ and $\sqcup$, though with increased complexity.

With these abstract definitions, the *validity* of variable-free constraints can be defined in a natural way. A variable-free goal G is valid if all assertions it contains are valid. An assertion $C_1 \vdash C_2$ is valid if the partial orderings in C_2 are entailed from C_1, using just the lattice properties of the relation $\leq$ and the variances of the various constructor arguments.

Example Let A, B, C be three distinct constants. Then $A \leq B \wedge B \leq C \vdash A \leq C$ is valid by the transitivity of $\leq$. Assertion $\vdash A \leq A \sqcup B$ is valid by the definition of join. Assertion $\vdash A \leq B$ is invalid: the empty assumption does not entail the conclusion.

3.3 Satisfiability

Validity as defined so far works for constraints without variables. When constraints mention variables, they are satisfiable if there exists a valuation of all variables such that the goal after value substitution is valid.

Satisfiability depends on the *ground terms* $\mathcal{T}$ that a variable can map into. Let $\mathcal{T}$ be the greatest fixed point of the following rules:

- All constants are in $\mathcal{T}$.
- $c(t_1, \ldots, t_{a(c)}) \in \mathcal{T}$ if $\forall_{i \in \{1,\ldots,a(c)\}}\; t_i \in \mathcal{T}$ and $c \in \mathbf{Con}$.

Notice that ground terms may be infinite. This feature is essential for modeling recursive types.

A valuation $\Phi : \mathbf{Var} \to \mathcal{T}$ is a function from variables to ground terms. A goal is *satisfiable* when there exists a valuation Φ such that the goal is valid after substitution using Φ.

Example Let $\alpha \in \mathbf{Var}$, $A, B, C \in \mathcal{T}$. Then $\vdash \alpha \leq A$ is trivially satisfiable by the valuation $\Phi(\alpha) = A$ or $\Phi(\alpha) = \bot$. However, $\vdash \alpha \leq A \wedge B \leq \alpha$ is unsatisfiable since otherwise, $B \leq A$ by the transitivity of $\leq$, yet this ordering on A and B is not entailed.

3.4 Expressiveness

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

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.

3.4.1 ML type inference

ML type inference maps naturally into constraint solving, since typing rules are usually equality constraints on types. Numerous efforts have been made in this direction (e.g., [2, 15, 17, 27, 37]).

Most of these formalizations are similar, so we discuss how Damas's Algorithm T [9] can be recast into our constraint language, extending the approach of Haack and Wells [15]. 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 5. However, our implementation does support a much larger set of language features, including `match` expressions and user-defined data types.

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

The typing rules that generate constraints are shown in Figure 5. Types t can be type variables to be inferred (α), the predefined integer type int, and function types constructed by $\to$.

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 $\mathtt{fn}\ x \to e$, 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 [13]. We leave that as future work.

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

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

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

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

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

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

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 $C = \{t_{1,1} = t'_1, \ldots, t_{1,n} = t'_n\}$

Figure 5. Constraint generation for a subset of ML. α and α_x are fresh variables in typing rules.

3.4.2 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 [10], 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 [31]. 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 [31] statically analyzes the security of information flow within programs. All types are annotated with security labels drawn from the decentralized label model (DLM) [30].

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.

Security policies on information are expressed as *labels* that mention these principals. For example, the confidentiality label {patient $\to$ 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.

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:

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

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

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. The proof can be found in the associated technical report [39].

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 $C_{\mathtt{b}} \leq$ {A $\to$ A} is added to the hypotheses of all constraints generated by the method body, where the constant $C_{\mathtt{b}}$ 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.

3.4.3 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:

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

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).

3.5 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 4). This graph is then analyzed using Bayesian principles to identify the explanations most likely to be correct (Section 5).

4. Constraint graph

The core constraint 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 context-free-language reachability in the graph.

4.1 Running example

We use the following example throughout this section to illustrate the key ideas behind the constraint graph representation.

Example Consider the following set of constraints.

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

We interpret $\leq$ here as the subtyping relation. The constructor $\mathsf{fn}(E_1, E_2)$ represents the function type $E_1 \to E_2$. 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 first assertion claims that α is a subtype of $\mathsf{fn}(\mathtt{ty1}, \mathtt{bool})$, with no hypotheses. The third assertion is similar. The second assertion says that β is a subtype of α under the assumption that `ty1` is a subtype of `ty2`.

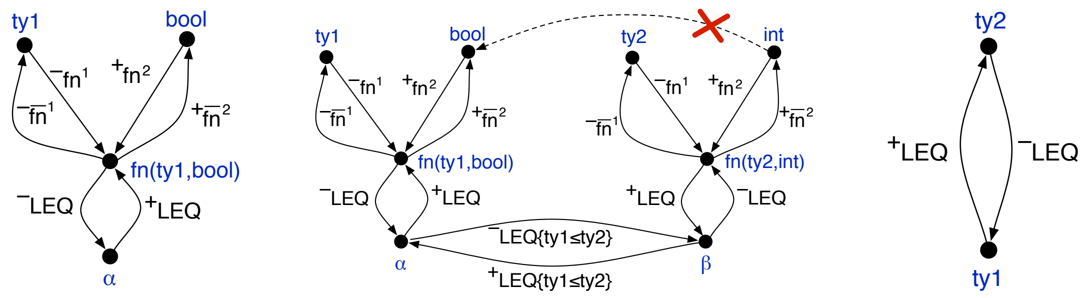
To determine whether this goal is satisfiable, we construct a constraint graph to infer partial orderings that must hold based on these constraints and the built-in, language-independent inference rules associated with the relation $\leq$, the constructors used, and the operators $\sqcup$ and $\sqcap$.

4.2 Constraint graph construction

The graph contains a node for each distinct element in the constraint system. For each partial ordering $E_1 \leq E_2$ appearing in assertion conclusions, a directed edge exists from E_1 to E_2, representing the legal flow of information. We call this edge an LEQ edge.

Hypotheses of assertions are recorded on the LEQ edges generated by the corresponding conclusions. We denote an edge annotated by hypothesis H as $\mathsf{LEQ}\{H\}$. For instance, the second constraint in our running example, $\mathtt{ty1} \leq \mathtt{ty2} \vdash \beta \leq \alpha$, generates an edge $\mathsf{LEQ}\{\mathtt{ty1} \leq \mathtt{ty2}\}$ from node β to node α.

Additional *constructor edges* in the constraint graph represent the action of constructors. Constructor edges connect the constructor’s arguments to the element representing its result. For example,



(a) Constructor edges (b) Full constraint graph (c) Hypothesis graph

Figure 6. Constraint graph generated from unsatisfiable constraints

there would be a constructor edge to the node representing the element $\text{fn}(\texttt{ty1}, \texttt{bool})$ from each of the nodes for `ty1` and `bool`, as illustrated in Figure 6(a).

Constructor edges include the following annotations: the constructor name, the argument position, and the *variance* of the parameter (covariant, contravariant or invariant). For instance, the edge labeled $(^-\text{fn}^1)$ connects the first argument to the constructor application. 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, $(^-\overline{\text{fn}}^1)$.

To simplify reasoning about the graph, LEQ edges are also duplicated in the reverse direction, with negative variance. Thus, the first assertion in the example, $\vdash \alpha \leq \text{fn}(\texttt{ty1}, \texttt{bool})$, generates a (^+LEQ) edge from α to $\text{fn}(\texttt{ty1}, \texttt{bool})$, and a (^-LEQ) edge in the other direction, as illustrated in Figure 6(a).

The constraint graph generated using all three assertions from the example is shown in Figure 6(b), excluding the dotted arrow.

Formal construction of the constraint graph Figure 7 formally presents a function $\mathcal{A}$ that translates 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**. The nodes of the constructed graph are implicitly defined by their connecting edges. Nodes are drawn from 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 4.

As shown, there are three kinds of edges. The LEQ edges, annotated with hypotheses, are generated by the translation rule for $\mathcal{A}[\![C \vdash E_1 \leq E_2]\!]$ and by the rules for meets and joins. Constructor edges are generated by the rules $\mathcal{E}[\![\mathit{cons}(E_1, \ldots, E_n)]\!]C$ and $\mathcal{E}[\![\overline{\mathit{cons}}^i(E)]\!]C$, which connect a constructor application to its arguments. Invariant arguments generate edges as though they were both covariant and contravariant, so twice as many edges are generated.

4.3 Inferring node orderings

The constraint graph facilitates inferring all $\leq$ relationships that can be proved using the corresponding constraints. The idea is to construct a context-free grammar, shown in Figure 8, whose productions correspond to inference rules for "$\leq$" 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 $\leq$ is expressed by the first grammar production, which derives $(^p\text{LEQ}\{H_1 \wedge H_2\})$ from consecutive LEQ edges $(^p\text{LEQ}\{H_1\})$ and $(^p\text{LEQ}\{H_2\})$, 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.

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

$$
\textbf{Graph} = \wp(\textbf{Edge}) \qquad \mathcal{A}[\![G]\!] : \textbf{Graph} \qquad \mathcal{E}[\![E]\!]C : \textbf{Graph}
$$

$$
\begin{aligned}
\mathcal{A}[\![A_1 \wedge \ldots \wedge A_n]\!] &= \bigcup_{i \in 1..n} \mathcal{A}[\![A_i]\!]\\
\mathcal{A}[\![C \vdash I_1 \wedge \ldots \wedge I_n]\!] &= \bigcup_{i \in 1..n} \mathcal{A}[\![C \vdash I_i]\!]\\
\mathcal{A}[\![C \vdash E_1 \leq E_2]\!] &= \mathcal{E}[\![E_1]\!]C \cup \mathcal{E}[\![E_2]\!]C\\
&\quad \cup \{(^+\text{LEQ})\{C\}(E_1 \mapsto E_2), (^-\text{LEQ})\{C\}(E_2 \mapsto E_1)\}\\
\mathcal{E}[\![\alpha]\!]C = \mathcal{E}[\![c]\!]C &= \mathcal{E}[\![\bot]\!]C = \mathcal{E}[\![\top]\!]C = \emptyset\\
\mathcal{E}[\![\mathit{cons}(E_1, \ldots, E_n)]\!]C &= \bigcup_{i \in 1..n} \Big(\{(^{p_i}\mathit{cons}^i)(E_i \mapsto \mathit{cons}(E_1, \ldots, E_n))\}\\
&\quad \cup \{(^{p_i}\overline{\mathit{cons}}^i)(\mathit{cons}(E_1, \ldots, E_n) \mapsto E_i)\} \cup \mathcal{E}[\![E_i]\!]C \Big)\\
\mathcal{E}[\![\overline{\mathit{cons}}^i(E)]\!]C &= \{(^{p_i}\mathit{cons}^i)(\overline{\mathit{cons}}^i(E) \mapsto E)\}\\
&\quad \cup \{(^{p_i}\overline{\mathit{cons}}^i)(E \mapsto \overline{\mathit{cons}}^i(E))\} \cup \mathcal{E}[\![E]\!]C
\end{aligned}
$$

(where p_i is the variance of argument i to constructor *cons*)

$$
\begin{aligned}
\mathcal{E}[\![E_1 \sqcup E_2]\!]C &= \bigcup_{i \in 1..2} \Big(\{(^+\text{LEQ})\{C\}(E_i \mapsto E_1 \sqcup E_2)\}\\
&\quad \cup \{(^-\text{LEQ})\{C\}(E_1 \sqcup E_2 \mapsto E_i)\} \cup \mathcal{E}[\![E_i]\!]C \Big)\\
\mathcal{E}[\![E_1 \sqcap E_2]\!]C &= \bigcup_{i \in 1..2} \Big(\{(^+\text{LEQ})\{C\}(E_1 \sqcap E_2 \mapsto E_i)\}\\
&\quad \cup \{(^-\text{LEQ})\{C\}(E_i \mapsto E_1 \sqcap E_2)\} \cup \mathcal{E}[\![E_i]\!]C \Big)
\end{aligned}
$$

Figure 7. Construction of the constraint graph

$$
\begin{aligned}
(^p\text{LEQ}\{H_1 \wedge H_2\}) &::= (^p\text{LEQ}\{H_1\})\ (^p\text{LEQ}\{H_2\})\\
(^+\text{LEQ}\{H\}) &::= (^pc^i)\ (^p\text{LEQ}\{H\})\ (^p\overline{c}^i)\\
(^-\text{LEQ}\{H\}) &::= (^pc^i)\ (^{\overline{p}}\text{LEQ}\{H\})\ (^p\overline{c}^i)
\end{aligned}
$$

where $c \in \textbf{Con}$, $1 \leq i \leq a(c)$, $p \in \{+, -\}$, $\overline{+} = -$ and $\overline{-} = +$

Figure 8. Context-free grammar for (^+LEQ) inference

The power of context-free grammars is needed in order to handle reasoning about constructors. In the running example, applying transitivity to the constraints yields $\texttt{ty1} \leq \texttt{ty2} \vdash \text{fn}(\texttt{ty2}, \texttt{int}) \leq \text{fn}(\texttt{ty1}, \texttt{bool})$. Then, because `fn` is contravariant in its first argument, we derive $\texttt{ty1} \leq \texttt{ty2}$. Similarly, we can derive $\texttt{int} \leq \texttt{bool}$, the dotted arrow in Figure 6(b).

To capture this kind of reasoning, we use the first two productions in Figure 8. In our example of Figure 6(b), the path from ty1 to ty2 has the following edges: $({}^{-}\mathsf{fn}^{1})$ $({}^{-}\mathsf{LEQ})$ $({}^{-}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ $({}^{-}\mathsf{LEQ})$ $({}^{-}\overline{\mathsf{fn}}^{1})$. These edges reduce via the first and then the second production to an edge $({}^{+}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ from ty1 to ty2. Note that the variance is flipped because the first constructor argument is contravariant. Similarly, we can infer another $({}^{+}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ edge from int to bool.

The third grammar production in Figure 8 is the dual of the second production, ensuring the invariant that each $({}^{+}\mathsf{LEQ})$ edge has an inverse $({}^{-}\mathsf{LEQ})$ edge. In our example of Figure 6(b), there is also an edge $({}^{-}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ from ty2 to ty1, derived from the following edges: $({}^{-}\mathsf{fn}^{1})$ $({}^{+}\mathsf{LEQ})$ $({}^{+}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ $({}^{+}\mathsf{LEQ})$ $({}^{-}\overline{\mathsf{fn}}^{1})$. These edges reduce via the first and then the third production to an edge $({}^{-}\mathsf{LEQ}\{\texttt{ty1} \leq \texttt{ty2}\})$ from ty2 to ty1.

Computing all inferable $({}^{+}\mathsf{LEQ})$ edges according to the context-free grammar in Figure 8 is an instance of *context-free-language reachability*, which is well-studied in the literature [5, 28] and has been used for a number of program-analysis applications [34]. We adapt the dynamic programming algorithm of Barrett et al. [5] to find shortest $({}^{+}\mathsf{LEQ})$ paths. We call such paths *supporting paths* since the hypotheses along these paths justify the inferred $({}^{+}\mathsf{LEQ})$ edges. We extend this algorithm to also handle join and meet nodes.

Take join nodes, for instance (meet is handled dually). The rule $E_1 \sqcup E_2 \leq E \iff E_1 \leq E \ \wedge \ E_2 \leq E$ can be used in two directions. The direction from left to right is already handled when we construct edges for join elements (Figure 7).

To use the rule in the other direction, we use the following procedure when a new edge $({}^{+}\mathsf{LEQ})\{C\}(n_1 \mapsto n_2)$ 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 $({}^{+}\mathsf{LEQ})$ edge to n_2.

4.4 Checking the satisfiability of $({}^{+}\mathsf{LEQ})$ edges

A $({}^{+}\mathsf{LEQ})$ edge, whether inferred or specified directly in an assertion, is added to the graph only when the corresponding $\leq$ ordering is entailed by the constraints along the supporting path. Hence, the constraints along the path must be unsatisfiable if the partial ordering on the end nodes is unsatisfiable.

When either end node of a $({}^{+}\mathsf{LEQ})$ edge is a variable or a $\sqcup(\sqcap)$ node where at least one argument of $\sqcup(\sqcap)$ is a variable, the edge is trivially satisfiable and hence not informative for error diagnosis. For simplicity, we ignore such edges and refer subsequently only to informative $({}^{+}\mathsf{LEQ})$ edges. Two informative $({}^{+}\mathsf{LEQ})$ edges can be inferred in Figure 6(b). These edges are $\texttt{int} \mapsto \texttt{bool}$ and $\texttt{ty1} \mapsto \texttt{ty2}$, though only the first is shown.

A $({}^{+}\mathsf{LEQ})$ edge holds only if all hypotheses on the edge hold too. Therefore, the satisfiability of an edge $({}^{+}\mathsf{LEQ})\{C\}(n_1 \mapsto n_2)$ is equivalent to the satisfiability of the assertion $C \vdash n_1 \leq n_2$. In our running example, the combined hypotheses along both informative edges are $\texttt{ty1} \leq \texttt{ty2}$. Therefore, satisfiability of the constraint system reduces to satisfiability of these assertions:

$$\texttt{ty1} \leq \texttt{ty2} \vdash \texttt{int} \leq \texttt{bool} \qquad \texttt{ty1} \leq \texttt{ty2} \vdash \texttt{ty1} \leq \texttt{ty2}$$

To check the satisfiability of these assertions, we test if the conclusion can be proved from all constraints in the hypothesis. Recall that a constraint graph facilitates the inference of all provable partial ordering given a set of constraints. Therefore, a *hypothesis graph* is constructed in exactly the same way as the constraint graph to find all provable $\leq$ relations.

Specifically, to test if an edge $({}^{+}\mathsf{LEQ})\{C\}(n_1 \mapsto n_2)$ is satisfiable, we construct a hypothesis graph using C as described in Section 4.2 and find all inferable $({}^{+}\mathsf{LEQ})$ edges as described in Section 4.3. Edge $({}^{+}\mathsf{LEQ})\{C\}(n_1 \mapsto n_2)$ is unsatisfiable if the relationship $({}^{+}\mathsf{LEQ})(n_1 \mapsto n_2)$ cannot be inferred from the hypothesis graph using C.

For our running example, the hypothesis graphs for both informative edges are the same. From this graph, shown in Figure 6(c), $\texttt{int} \leq \texttt{bool}$ is not provable. The constraints along the supporting path from int to bool form a proof of unsatisfiability.

Satisfiable and unsatisfiable paths 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 $({}^{+}\mathsf{LEQ})$ 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.

5. Ranking explanations

The algorithm in Section 4 identifies unsatisfiable paths in the constraint graph, which correspond to sets of unsatisfiable constraints expressed by our constraint language.

Although the information along unsatisfiable paths already captures why the 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.

5.1 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 $o_i \in \{\texttt{unsat}, \texttt{sat}\}$ 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 leave the incorporation of language-specific knowledge to future work.

We also assume the probability of each entity being the cause is independent.[1] 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 5.3.

$P(o|E,H)$ 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. 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 $P(o_i = \texttt{unsat}|E,H)$ is equal to 1 in this case, and is otherwise 0.
2. A satisfiable path is unlikely (with some constant probability $P_2 < 0.5$) to contain a wrong entity. Since adding or removing H does not affect a path that is already satisfiable, $P(o_i = \texttt{sat}|E,H)$ is not affected by H. Hence, we have $P(o_i = \texttt{sat}|E,H) = P_2$ if path p_i contains a constraint generated by some entity in E. Otherwise, $P(o_i = \texttt{sat}|E,H) = 1 - P_2$.

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 $o_i = \texttt{unsat}$ by heuristic 1). We denote this set by $\mathcal{G}$. Suppose nsat (a constant) paths are satisfiable, and entities E appear on k_E of them. Then, based on the simplifying assumptions made, we have

$$\begin{aligned} &\arg\max_{E\subseteq\Omega, H\subseteq\Psi} P_\Omega(E) P_\Psi(H) P(o|E,H) \\ =& \arg\max_{E\subseteq\Omega, H\subseteq\Psi} P_1^{|E|} P_\Psi(H) P_2^{k_E} (1-P_2)^{\texttt{nsat}-k_E} \Pi_{i:o_i=\texttt{unsat}} P(o_i|E) \\ =& \arg\max_{(E,H)\in\mathcal{G}} P_1^{|E|} (P_2/(1-P_2))^{k_E} P_\Psi(H) \end{aligned}$$

An intuitive understanding of this estimation is that the cause must explain all unsatisfiable paths; the wrong entities are likely to be small ($|E|$ is small) and not used often on satisfiable paths (since $P_2 < 1 - P_2$ by heuristic 2); the missing hypothesis is likely to be weak and small, as defined in Section 5.3, which maximizes the term $P_\Psi(H)$.

Although this estimation is affected by the values of P_1 and P_2, empirical study suggests that the diagnosis result is insensitive to their values across a broad range (see Section 6.2.1).

5.2 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 [16], based on novel heuristics, to calculate optimal solutions in a practical way.

[1] 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.

A* search is a heuristic search algorithm for finding minimum-cost solution nodes in a graph of search nodes. In our instance of the algorithm, 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 $\mathcal{P}_{\texttt{rm}}$, since C_1 is usually larger than C_2. More specifically, $h(n) = 0$ if $\mathcal{P}_{\texttt{rm}} = \emptyset$. Otherwise, $h(n) = C_1$ if $\mathcal{P}_{\texttt{rm}}$ 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. The proof is included in the associated technical report [39]. The heuristic search algorithm is also efficient in practice: on current hardware, it takes about 10 seconds when the search space is over 2^{1000}. More performance details are given in Section 6.

Since the remaining part of our instance of A* search is largely standard, we leave the details in the accompanied technical report [39]. The only nonstandard feature is that the search stops when a suboptimal suggestion is found, rather than when the first suggestion is found, since we are interested in all top-ranked suggestions.

5.3 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.

5.3.1 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.

5.3.2 Missing hypothesis

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

We define a *missing hypothesis* as follows:

DEFINITION 1. *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 $\forall P_i \in \mathcal{P} \,.\, \mathcal{H}(P_i) \wedge \bigwedge_{I \in S} I \vdash \mathcal{C}(P_i)$.*

Intuitively, adding all inequalities in the missing hypothesis to the assertions' hypotheses removes all unsatisfiable paths in the constraint graph.[2]

Example Returning to the example in Section 5.3.1, it is easy to verify that $\texttt{Alice} \leq \texttt{Bob}$ is a missing hypothesis that makes all of the assertions valid.

5.3.3 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: $\mathcal{C}(\mathcal{P}) = \bigcup\{\mathcal{C}(P_i) \mid P_i \in \mathcal{P}\}$.

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

DEFINITION 2. *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 $\mathcal{C}(\mathcal{P})$ that forms a missing hypothesis is maximally weak:

LEMMA 1. *$\forall S \subseteq \mathcal{C}(\mathcal{P})$. S is a missing hypothesis $\implies$ no missing hypothesis is strictly weaker than S.*

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

The lemma above suggests that subsets of $\mathcal{C}(\mathcal{P})$ may be good candidates for a weak missing hypothesis. However, they are not necessarily minimal. For instance, the entire set $\mathcal{C}(\mathcal{P})$ 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 $c_j = \mathcal{C}(P_j)$ 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 5.3.1. The missing hypothesis $\texttt{Alice} \leq \texttt{Bob}$ is both the weakest and minimal.

Based on Lemma 1 and the definition above, finding a minimal weakest missing hypothesis in $\mathcal{C}(\mathcal{P})$ is equivalent to finding the minimum subset of $\mathcal{C}(\mathcal{P})$ which subsumes all $c \in \mathcal{C}(\mathcal{P})$. This gives us the following algorithm:

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

1. Construct the set $\mathcal{C}(\mathcal{P})$ from the unsatisfiable paths.
2. For all c_i, c_j in $\mathcal{C}(\mathcal{P})$, 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 $\mathcal{C}(\mathcal{P})$, 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.

6. Evaluation

6.1 Implementation

We implemented our general error diagnostic tool in Java. The implementation includes about 5,500 lines of source code, excluding comments and blank lines.

As input, the diagnostic tool reads in constraints following the syntax of Figure 4. The program analyses to be diagnosed must be modified to emit those constraints.

To evaluate our error diagnostic tool on real-world program analyses, we modified the Jif compiler and an extension to the OCaml compiler, EasyOCaml [12], to generate constraints in our constraint language format. EasyOCaml is an extension of OCaml 3.10.2 that generates the labeled constraints defined in [15].

Generating constraints in our language format involved only modest effort. Changes to the Jif compiler include about 300 LoC (lines of code) above more than 45,000 LoC in the Jif 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 to trace constraints back to the corresponding source code.

6.2 Case study: OCaml error reporting

To evaluate the quality of our ranking algorithm, we used a corpus of previously collected OCaml programs containing errors, collected by Lerner et al. [23]. 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.

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

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.

Analysis 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.

6.2.1 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 5.2). 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 P_1 and P_2, we collect two important statistics for a wide range of P_1 and P_2 values: 1) the number of programs where the actual error is missing in top rank suggestions (among 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 for any $0 < P_2 < 0.5$, P_1 decreases exponentially from left to right. The last column corresponds to the special case when $P_2 = 0.5$.

Empirically, the overall suggestion quality is best when $P_1 = P_2'^3$, where $P_2' = P_2/1 - P_2$. However, the quality of the suggestions is close for any P_1 and P_2 s.t. $P_2'^2 \leq P_1 \leq P_2'^6$; the results are not very sensitive to the choice of these parameters.

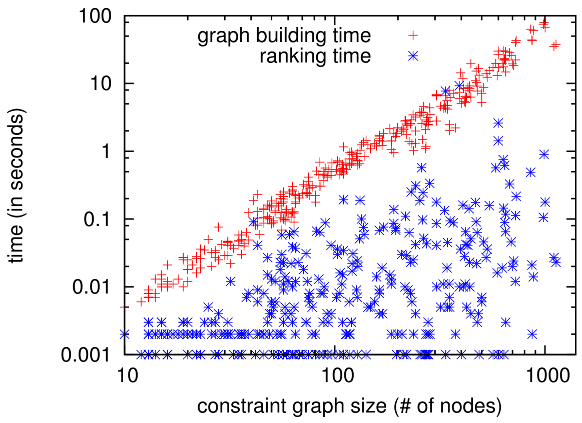


Figure 10. Performance

If satisfiable paths are ignored ($P_2 = 0.5$, that is, $C_2 = 0$), the top-rank suggestion size is much larger, and more errors are missing. Hence, using satisfiable paths is important to suggestion quality.

The quality of the error report is also considerably worse when P_1 is very large relative to P_2 ($P_1 = P_2'$). This result shows that unsuccessful paths are more important than successful paths, but that ascribing too importance to the unsuccessful paths (e.g., at $P_1 = P_2'^{10}$) also hurts the quality of the error report.

6.2.2 Comparison with OCaml and Seminal

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 [23], 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 9:

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.

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.

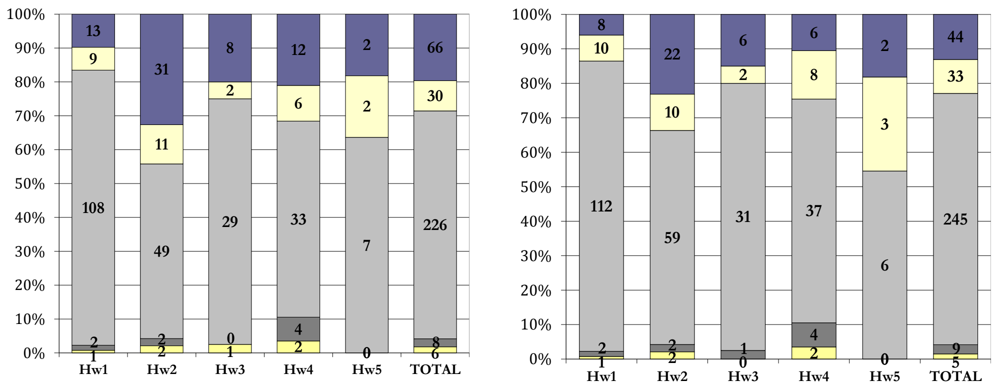
Compared with both OCaml and Seminal, our tool consistently identifies a higher percentage of error locations across all homeworks, with an average of 96%.

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.

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

	$P_1 = P'_2$	$P_1 = P'^2_2$	$P_1 = P'^3_2$	$P_1 = P'^4_2$	$P_1 = P'^5_2$	$P_1 = P'^6_2$	$P_1 = P'^{10}_2$	$P_2 = 0.5$
Missed Error	21	15	14	17	17	16	22	23
Avg. Sugg. Size	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 values of P_1 and P_2, where $P'_2 = P_2/1 - P_2$.



(a) Comparison with the OCaml compiler (b) Comparison with Seminal

Figure 9. 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.

the programmer made a similar error multiple times. Our tool fails to identify such errors because they 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.

6.2.3 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 10. 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.

6.3 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. 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 2. 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. 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 2:

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.

	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 3. 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

6.4 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 [3]. 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 3.

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 6.2 and 6.3, where error types are known ahead.

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 is the most promising approach, 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.

7. 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 [13]. 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 [34]. Melski and Reps [28] 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 [36] and of Johnson and Walz [19]. 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 [17] provides more details.

In the first direction, several efforts [8, 13, 15, 33, 35] improve the basic idea of Wand [36] in various ways. Despite the attractiveness of feeding a full explanation to the programmer, the reports are usually verbose and hard to follow [17].

In the second direction, one approach is to alter the order of type unification [22, 26]. But since the error location may be used anywhere during the unification procedure, any specific order fails in some circumstance. Some prior work [17, 19] builds a type graph from a more limited constraint language and infers error locations based on heuristics mostly tailored for type inference. Though the "weighted options" heuristic in [19] uses successful type unifications to distinguish abnormal types from normal ones, information about satisfiable paths is leveraged with finer-granularity in our approach, to distinguish the *constraints* that caused errors. This is shown to be effective in Section 6.2.1.

A third approach is to generate fixes for errors by searching for similar programs [23, 27] or type substitutions [7] that do type-check. Unfortunately, we cannot obtain a common corpus to perform direct comparison with some of this prior work [7, 27]. It is worth noting that the ranking heuristics used in [7] are language-specific: there is no obvious way to extend them to information flow, for instance. We are able to compare directly with the work of Lerner et al. [23]; the results of Section 6.2 suggest that our approach finds error locations more accurately. In fact, by pinpointing where searches for fixes are likely to be productive, our approach ought to be complementary.

For information-flow control, King et al. [20] 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. [38] 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 [38] 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 [21, 25]. 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., [24, 40]). 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. 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 [4]. 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. on error diagnosis using abductive inference [11]. 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 algorithm in [11] infers missing hypotheses for a single assertion, while our tool finds missing hypotheses that satisfy a *set* of assertions. Further, the algorithm of [11] 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* (e.g., `Alice` $\leq$ `Bob` in Section 5.3.1).

Recent work by Blackshear and Lahiri [6] 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 above to Dillig et al. [11], our algorithm also infers missing partial orderings on constructors, not just additional specifications on variables.

8. Conclusion

Better tools for helping programmers locate the errors detected by program analysis should make them more willing to use the many powerful program analyses that have been developed. The science of diagnosing programmer errors is still rather primitive, but this paper 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 two very different languages, OCaml and Jif, with little 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 two very different type systems, it would likely be fruitful to apply these ideas to other type systems and program analyses, and to explore more sophisticated ways to estimate the likelihood of different error explanations.

Acknowledgments

Nate Foster, Mike George, Chinawat Isradisaikul, Jean-Baptiste Jeannin, Vincent Rahli, Robert Soulé and Ross Tate gave many useful comments on this presentation. We also thank Ben Lerner and Dan Grossman for making available the excellent data set they used for their work on Seminal, Dexter Kozen for pointing out the similarity between our constraints and set constraints, and Mike George for a well-organized and labeled set of Jif test cases.

This work was supported by two grants from the Office of Naval Research, N00014-09-1-0652 and N00014-13-1-0089, by MURI grant FA9550-12-1-0400, by a grant from the National Science Foundation (CCF-0964409), and by a grant administered by the Air Force Research Laboratory.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.
[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Conf. Functional Programming Languages and Computer Architecture*, pp. 31–41, 1993.
[3] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE Symp. on Security and Privacy*, pp. 191–205, May 2012.
[4] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 30*, pp. 97–105, Jan. 2003.
[5] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30:809–837, 2000.
[6] S. Blackshear and S. K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI'97*, pp. 209–218, 2013.
[7] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL 41*, Jan. 2014.
[8] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical report, Indiana University, December 1995.
[9] L. M. M. Damas. *Type assignment in programming languages*. PhD thesis, Department of Computer Science, University of Edinburgh, 1985.
[10] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.
[11] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI'12*, pp. 181–192, 2012.
[12] EasyOCaml. http://easyocaml.forge.ocamlcore.org.
[13] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Prog. Lang. Syst.*, 28(6):1035–1087, Nov. 2006.
[14] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, 2nd edition, 2004.
[15] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1–3):189–224, 2004.
[16] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.
[17] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.
[18] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.
[19] G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL 13*, pp. 44–57, 1986.
[20] D. King, T. Jaeger, S. Jha, and S. A. Seshia. Effective blame for information-flow violations. In *Int'l Symp. on Foundations of Software Engineering*, pp. 250–260, 2008.
[21] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI'06*, pp. 161–176, 2006.
[22] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Prog. Lang. Syst.*, 20(4):707–723, 1998.
[23] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI'07*, pp. 425–434, 2007.
[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI'05*, pp. 15–26, 2005.
[25] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI'09*, pp. 75–86, 2009.
[26] B. J. McAdam. On the unification of substitutions in type inference. In *Implementation of Functional Languages*, pp. 139–154, 1998.
[27] B. J. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, 2001.
[28] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2):29–98, 2000.
[29] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
[30] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP'97*, pp. 129–142, 1997.
[31] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, `www.cs.cornell.edu/jif`, July 2006.
[32] OCaml programming language. http://ocaml.org.
[33] V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical Report HW-MACS-TR-0079, Heriot-Watt university, 2010.

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

[35] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, 2001.

[36] M. Wand. Finding the source of type errors. In *POPL 13*, 1986.

[37] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[38] J. Weijers, J. Hage, and S. Holdermans. Security type error diagnosis for higher-order, polymorphic languages. In *ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 3–12, 2013.

[39] D. Zhang and A. C. Myers. Toward general diagnosis of static errors: Technical report. Technical Report http://hdl.handle.net/1813/33742, Cornell University, Aug. 2014.

[40] A. X. Zheng, B. Liblit, and M. Naik. Statistical debugging: simultaneous identification of multiple bugs. In *ICML'06*, pp. 1105–1112, 2006.