# Diagnosing Haskell Type Errors

Danfeng Zhang[1], Andrew C. Myers[1], Dimitrios Vytiniotis[2] and Simon Peyton-Jones[2]

[1]Department of Computer Science, Cornell University
{zhangdf,andru}@cs.cornell.edu
[2]Microsoft Research Cambridge
{dimitris,simonpj}@microsoft.com

April 12, 2015

**Abstract**

Type inference engines often give terrible error messages, and the more sophisticated the type system the worse the problem. We show that even with highly expressive type system implemented by the Glasgow Haskell Compiler (GHC)—including type classes, GADTs, and type families—it is possible to identify the *most likely source* of the type error, rather than the *first source* that the inference engine trips over. To determine which are the likely error sources, we apply a simple Bayesian model to a graph representation of the typing constraints; the satisfiability or unsatisfiability of paths within the graph provides evidence for or against possible explanations. While we build on prior work on error diagnosis for simpler type systems, inference in the richer type system of Haskell requires extending the graph with new nodes. The augmentation of the graph creates challenges both for Bayesian reasoning and for ensuring termination. Using a large corpus of Haskell programs, we show that this error localization technique is practical and significantly improves accuracy over the state of the art.

## 1 Introduction

Type systems and other static analyses help reduce the need for debugging at run time, but sophisticated type systems can lead to terrible error messages. The difficulty of understanding these error messages interferes with the adoption of expressive type systems.

Even for program errors that are detected statically, It can be difficult to determine where the mistake lies in the program. 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.

Recent work by [36] made progress on this problem, demonstrating that a more holistic Bayesian approach to localizing errors can improve accuracy significantly, for at least some nontrivial type systems (OCaml and Jif). A key idea of that work is to represent constraints as a *constraint graph* that allows efficient reasoning about a possibly large number of counterfactual error explanations.

However, that graph representation cannot handle richer type systems in which the reasoning process requires a constraint solver that can handle quantified propositions involving functions over types. Type classes and type families, as supported by GHC [21], require such a solver, whereas simple polymorphic types as in ML do not [28, 8]. Better error localization would be very valuable for such type systems, because their error messages can be particularly inscrutable. In the constraint graph representation, however, a solver

for such rich type systems needs to add *new nodes* to the constraint graph, posing challenges for soundness, completeness, termination, and efficiency of the analysis.

Our principal contribution is to show that an approach based on Bayesian reasoning can be applied even to such type systems. Specifically:

- We define a constraint language and constraint graph representation that can encode a broad range of type systems and other analyses. In particular, they add the ability to handle the features of the expressive type system of Haskell, including type classes, GADTs, and type families. (§3 and §4)
- We extend the constraint-graph solving technique of [36] to allow 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 terminiates. (§5)
- We develop a Bayesian model for programmer mistakes that accounts for the richer representation of constraints and the presence of derived constraints. (§6)
- We have implemented this technique as an extension to the publicly available SHErrLoc diagnostic tool [30], using GHC itself as the constraint generator so that we handle all of Haskell. (§7)
- Using a corpus of more than 300 Haskell programs, many written by students solving programming assignments, we show that mistakes are more accurately located than with prior techniques. Further, the performance of the diagnostic algorithm is acceptable. (§8)

## 2 The challenge we tackle

Type inference problems can generally be expressed in terms of solving a set of constraints on type expressions, and type inference succeeds when variables in the constraints can be assigned types that make all the constraints satisfiable.

When constraints are unsatisfiable, the question is how to *identify the program point* that is most likely to be the error source. The standard practice is to report the program point that generates the last failed constraint. Unfortunately, this simple approach often results in misleading error messages—the actual error source may be far from that program point.

As a motivating example, consider the following Haskell program from [18], which fails to type-check:

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

The actual mistake is that the second equality test (==, in line 2) should be subtraction (-), 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 most type checkers, GHC included, implement 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.

Rather than reporting the location of a *single* failed constraint, we might think to report *all* locations that might contribute to the error (e.g., as in [35, 7, 32, 10]). But such error reports are often verbose and hard to understand [14], because many expressions can be at least partly involved in a given failure.

A more promising approach is described by [36], where the structure of the constraint system *as a whole* is analyzed, reporting the most likely error rather than the error first encountered. The question we address

in this paper is: *can this holistic approach be scaled to handle type systems more sophisticated than those of ML and Jif?*

**Haskell features we tackle** Haskell is recognised as having a particularly rich type system, and hence makes an excellent test case. Besides type classes, we treat these features:

- **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 [27]. For instance, the program below

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

  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. For example, this program

```
f x = let g :: forall a. a -> (a,a)
          g z = (z,x)
      in (g 42, g True)
```

  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 [27].

- **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.

**Our approach** We develop a rich *constraint language* (§3) that can encode all type constraints generated by GHC [33]. We use GHC itself to generate type constraints for Haskell programs with all sophisticated features above, and then translate these constraints to our constraint language. To simplify the presentation, we collapse these two steps into one, by generating constraints in our language directly. Our tool handles all GHC constraints but for illustration we use a sufficiently rich Haskell subset (§4).

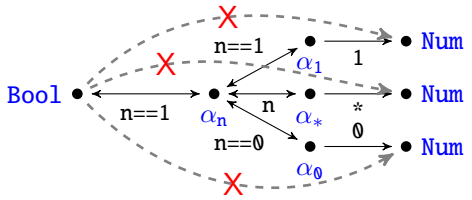



Figure 1: Part of the graph for the Haskell example.

The set of constraints is then transformed into a *constraint graph* (§5.1). For example, part of the graph for the motivating (factorial) example is depicted in Figure 1, where nodes $\alpha_{\texttt{n}}$, $\alpha_{\texttt{0}}$, $\alpha_{\texttt{1}}$ and $\alpha_{*}$ represent the types of `n`, `0`, `1` and the first parameter of `*` respectively, and each bidirectional edge represents type equality between the end nodes. In this figure, each edge is annotated with the expression that generates it. For example, the edge between $\alpha_{\texttt{n}}$ and **`Bool`** is generated since the return type of `(n == 1)`, namely **`Bool`**, must equal the type of `n`, the argument of `fac`.

Besides equality constraints, Haskell also generates *type class* constraints. Type classes introduce, in effect, relations over types. For example, the type of literal `0` can be any instance of the type class **`Num`**, such as **`Int`** and **`Float`**. We use a *directed edge*, encoding a partial ordering, to express these constraints. For example, the edge from $\alpha_{\texttt{0}}$ to **`Num`** in Figure 1 means that $\alpha_{\texttt{0}}$ must be an instance of **`Num`**.

The constraint graph is then *saturated and expanded* so that all possible deductions are represented as graph edges (§5.2). For example, the dashed edges in Figure 1 are derived by transitivity.

Each edge in the saturated graph is then *classified* as satisfiable or unsatisfiable (§5.3). For example, the edges marked with a red X are unsatisfiable, since **`Bool`** is not an instance of **`Num`**.

Finally, we use the classification of constraint edges to assign the most likely error source, according to Bayesian principles (§6). Taken together, the chosen error locations should: 1) explain all unsatisfiable paths, 2) be small, and 3) not appear often on satisfiable paths. In accordance with these three principles, we correctly determine expression `(n == 1)` to be the most likely cause of the error.

**What is new** The general plan of graph generation, saturation, and classification follows [36]. The new aspects are these: first, a rich constraint language that can encode the expressive type system of Haskell (§3), including type class constraints; second, the encoding of type class constraints as inequalities (§4.2); third, a new graph-saturation algorithm, which handles type classes and type families by generating new nodes and edges in the constraint graph (§5.2); fourth, an edge-classification algorithm that correctly handles nested quantifiers (§5.3); finally, a modified Bayesian model that takes the creation of new nodes and edges into account (§6).

# 3 The SCL constraint language

We substantially modified and extended the constraint language of [36] in order to handle the rich type system of Haskell. The most significant new features of the new constraint language are quantified axioms, nested universally and existentially quantified variables, and type-level functions.

## 3.1 Syntax of the SCL constraint language

Figure 2 presents the syntax of the new constraint language, which we call SCL. A top-level goal $G$ 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 the assumption $H$.

| | | | |
|---|---|---|---|
| Unification variables | $\alpha, \beta, \gamma$ | Constructors | $\mathit{con}$ |
| Skolem variables | $\mathsf{a}, \mathsf{b}, \mathsf{c}$ | Functions | $\mathit{fun}$ |
| Quantified variables in hypothesis | | | $a, b, c$ |

$$G ::= A_1 \wedge ... \wedge A_n \ ^{(n \geq 0)} \qquad A ::= H \vdash I$$
$$H ::= Q_1 \wedge ... \wedge Q_n \ ^{(n \geq 0)} \qquad Q ::= \forall \overline{a} \,.\, C \Rightarrow I$$
$$C ::= I_1 \wedge ... \wedge I_n \ ^{(n \geq 0)} \qquad I ::= E_1 \leq E_2$$
$$E ::= \alpha_\ell \mid \mathsf{a}_\ell \mid a \mid \mathit{con}\ \overline{E} \mid \mathit{fun}\ \overline{E}$$

Figure 2: Syntax of SCL constraints.

**Constraints** A constraint $C$ is a conjunction of inequalities $E_1 \leq E_2$ over elements from the constraint element domain $E$ (typically types of the source language), where $\leq$ defines a partial ordering on elements[1]. 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.

**Quantified axioms in hypotheses** As [36] do, 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 ($\mathsf{a}, \mathsf{b}, \ldots$) 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 $\mathsf{a}_1 = \mathsf{b}_1 \vdash (\mathsf{a}_1, \mathsf{b}_1) = \alpha_2$ to represent $\forall \mathsf{a}, \mathsf{b}. \exists \alpha.\ \mathsf{a} = \mathsf{b} \vdash (\mathsf{a}, \mathsf{b}) = \alpha$. Any assertion written using quantifiers can be put into prenex normal form and therefore can represented using level numbers.

**Constructors and functions over constraint elements** As well as variables, an element $E$ may be an application $\mathit{con}\ \overline{E}$ of a type constructor $\mathit{con} \in \mathbf{Con}$, or an application $\mathit{fun}\ \overline{E}$ of a type-function $\mathit{fun} \in \mathbf{Fun}$. Constants are nullary constructors, with arity 0. Since constructors and functions are global, no levels are associated with them. Our full constraint language and implementation support contravariant and invariant constructors as well, but in order to keep this paper focused on the key challenges and contributions, we assume all constructors are covariant hereafter.

The main difference between a type constructor $\mathit{con}$ and a type function $\mathit{fun}$ is that functions are not necessarily injective (i.e., $\mathit{fun}\ \overline{\tau} = \mathit{fun}\ \overline{\tau'} \not\Rightarrow \overline{\tau} = \overline{\tau'}$), but constructors can be decomposed (i.e., $\mathit{con}\ \overline{\tau} = \mathit{con}\ \overline{\tau'} \Rightarrow \overline{\tau} = \overline{\tau'}$)

## 3.2 Validity and satisfiability

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

[1]The full constraint language also supports lattice joins and meets on elements. We omit them here since 1) they are not needed to represent Haskell constraints, and 2) adding them is straightforward.

$$\text{(CONJ)}\ \frac{H \vdash C_1 \quad H \vdash C_2}{H \vdash C_1 \wedge C_2} \qquad (\leq \text{REF})\ \frac{}{H \vdash E \leq E} \qquad (\leq \text{TRAN})\ \frac{H \vdash E_1 \leq E_2 \quad H \vdash E_2 \leq E_3}{H \vdash E_1 \leq E_3}$$

$$\text{(AXIOM)}\ \frac{\theta = [\overline{a \mapsto E}] \quad H \vdash \theta[C_1]}{H \wedge (\forall \overline{a}\ .\ C_1 \Rightarrow C_2) \vdash \theta[C_2]} \qquad \text{(DCOMP)}\ \frac{H \vdash E_i \leq E'_i \qquad 1 \leq i \leq a(c)}{H \vdash \mathit{con}(E_1, \ldots, E_{a(c)}) \leq \mathit{con}(E'_1, \ldots, E'_{a(c)})}$$

$$\text{(DECOMP)}\ \frac{H \vdash \mathit{con}(E_1, \ldots, E_{a(c)}) \leq \mathit{con}(E'_1, \ldots, E'_{a(c)})}{H \vdash E_1 \leq E'_1 \wedge \ldots \wedge E_{a(c)} \leq E'_{a(c)}}$$

$$\text{(FCOMP)}\ \frac{H \vdash E_i = E'_i \qquad 1 \leq i \leq f(c)}{H \vdash \mathit{fun}(E_1, \ldots, E_{f(c)}) = \mathit{fun}(E'_1, \ldots, E'_{f(c)})}$$

Figure 3: Entailment rules

A substitution $\theta$ is level-respecting if the substitution is well-scoped. More formally, $\forall \alpha_l \in dom(\theta), \mathsf{a}_m \in \mathit{fvs}(\theta[\alpha_l]).m \leq l$. 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 $\alpha$ with $\mathsf{b}$.

A unification-variable-free assertion $H \vdash I$ is *valid* if $I$ is entailed by $H$. A variable-free goal G is valid if all assertions it contains are valid.

The entailment rules (Figure 3) are entirely standard. Rule (AXIOM) instantiates a (potentially) quantified axiom in the following way: for any substitution $\theta$ that maps quantified variables $\alpha$ to constraint elements $E$, substituted constraints $\theta[C_2]$ are entailed whenever $H \vdash \theta[C_1]$. For example, the following assertion is valid by rule ($\leq$ REF) and (AXIOM) (substitute $\alpha$ 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 relationship already stated in the axioms. For example, $A \leq B \vdash A \leq B$ is (trivially) valid.

# 4 Generating constraints from a type system

The SCL constraint language is powerful enough to express advanced type system features in GHC. We demonstrate this constructively, by giving an algorithm to generate suitable constraints directly from a Haskell-like program.

## 4.1 Syntax

Figure 4 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 ($\sigma$) that may be polymorphic. For example,
  `let id :: (`$\forall a\ .\ a \to a$`) = (`$\lambda$`x.x) in ...`

  declares an identity function with a polymorphic type.

- A polymorphic type $\sigma$ 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

| | | | | |
|---|---|---|---|---|
| Term variables | | $x, y, z$ | Type classes | $D$ |
| Type variables | | $a, b, c$ | Type families | $F$ |
| Expressions | $e$ | $::= x \mid \lambda x\,.\,e \mid e_1\ e_2$ $\mid \texttt{let}\ x :: \sigma = e_1\ \texttt{in}\ e_2$ | | |
| Constraints | $P$ | $::= P_1 \wedge P_2 \mid \tau_1 = \tau_2 \mid D\ \overline{\tau}$ | | |
| Signatures | $\sigma$ | $::= \forall \overline{a}\,.\,P \Rightarrow \tau$ | | |
| Monotypes | $\tau$ | $::= a \mid \mathsf{Int} \mid \mathsf{Bool} \mid [\tau] \mid \mathsf{T}\ \overline{\tau} \mid F\ \overline{\tau}$ | | |
| Axiom schemes | $\mathcal{Q}$ | $::= P \mid \mathcal{Q}_1 \wedge \mathcal{Q}_2 \mid \forall \overline{a}\,.\,P \Rightarrow D\ \overline{\tau} \mid$ $\forall \overline{a}\,.\,F\ \overline{\tau} = \tau'$ | | |

Figure 4: Syntax of a Haskell-like language.

classes. The constraints in type signatures are subsumed by SCL, as we see shortly.

- The language supports type families: the syntax of types $\tau$ includes type families ($F\ \overline{\tau}$). A type can also be quantified type variables ($a$) and regular types ($\mathsf{Int}$, $\mathsf{Bool}$, $[\tau]$) that are no different from some arbitrary data constructor $\mathsf{T}$.

- An axiom scheme ($\mathcal{Q}$) 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 $\mathcal{Q}$:
  ```
  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 [34].

## 4.2 Constraint generation

Following prior work on constraint-based type inference [28, 25, 33], we formalize type inference as constraint solving, generating SCL constraints using the algorithm in Figure 5.

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 $\Gamma$, we may infer that an expression $e$ has a type $\tau$ and generates assertions $G$". The level $\ell$ 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 $\mathcal{Q}$, under the translation below.

Constraint translation $[\![P]\!] : C$

$$[\![D\ \tau]\!] := \tau \leq \mathsf{D} \qquad [\![D\ \overline{\tau}]\!] := (\mathsf{tup}_n\ \overline{\tau}) \leq \mathsf{D}$$
$$[\![P_1 \wedge P_2]\!] := [\![P_1]\!] \wedge [\![P_2]\!] \qquad [\![\tau_1 = \tau_2]\!] := (\tau_1 = \tau_2)$$

Type inference rules $H; \Gamma \models_\ell e : \tau \rightsquigarrow G$

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

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

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

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

Figure 5: Constraint generation.

**Type classes** How can we encode Haskell's type classes in SCL constraints? The encoding is shown in Figure 5, where we express a class constraint $D\ \tau$ as an inequality $\tau \leq \mathsf{D}$, where $\mathsf{D}$ is a unique constant for class $D$. The intuition is that $\tau$ is a member of the set of instances of $\mathsf{D}$. For a multi-parameter type class, the same idea applies, except that we use a constructor $\mathsf{tup}_n$ 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, $[\![\mathsf{Mul}\ \tau_1\ \tau_2\ \tau_3]\!] = (\mathsf{tup}_3\ \tau_1\ \tau_2\ \tau_3 \leq \mathsf{Mul})$.

### 4.3 Running example

We use the program in Figure 6 as a running example for the rest of this paper. 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 in Figure 6. The highlighted constraints are generated for the expression (`g ['a']`) in the following ways. Rule (VarCon) instantiates $d$ in the signature of `g` at type $\delta_0$, and generates the third constraint (recall that $(\mathsf{Num}\ \delta_0)$ is encoded as $(\delta_0 \leq \mathsf{Num})$). Instantiate the type of character `'a'` at type $\alpha_0$; hence $\alpha_0 = \mathsf{Char}$. Finally, using (App) on the call (`g ['a']`) generates a fresh type variable $\gamma_0$ and the fifth constraint $([\alpha_0] \to \gamma_0) = (\delta_0 \to \mathsf{Bool})$.

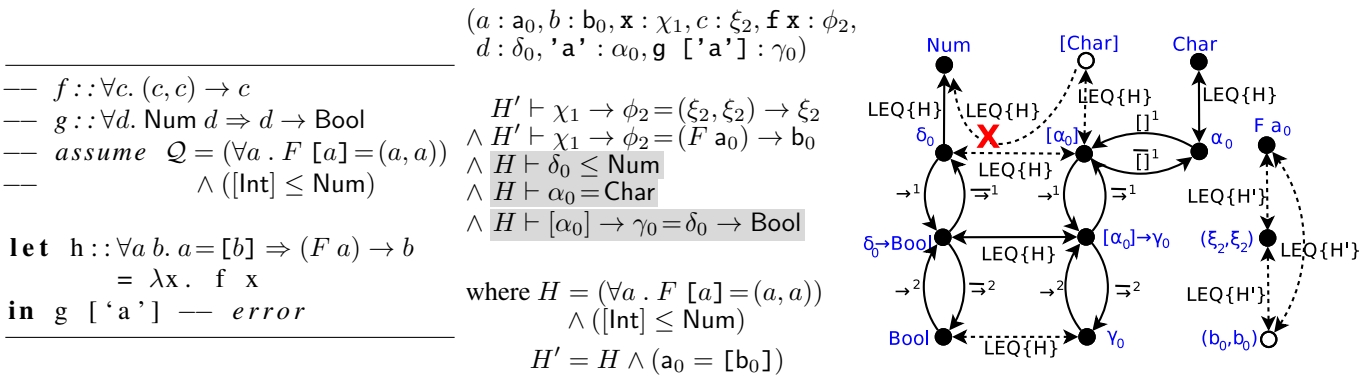



Figure 6: Running example. From left to right: program, generated constraints, part of the graph for constraints.

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 $a_0 = [b_0]$, added by rule (SIG).

# 5 Graph-based constraint analysis

[36] show that error report quality can be considerably improved by analyzing both satisfiable and unsatisfiable subsets of constraints. The key idea is analyze constraints in a graph representation. We present a novel algorithm that differs from this prior work in two significant ways: first, it rejects ill-typed programs that are accepted by the previous algorithm and accepts well-typed programs that are rejected by the previous algorithm; second, it supports the challenging language features discussed in §2.

## 5.1 Graph generation

A constraint graph is generated from assertions $G$ as follows. As a running example, Figure 6, excluding the white node and dotted edges, shows part of the generated constraint graph for the constraints in the centre column of the same figure.

1. For each assertion $H \vdash E_1 \leq E_2$, create 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 $(\mathit{con}\ \overline{E})$ in the graph, create a node for each of its immediate sub-elements $E_i$ (if they do not already exist); add a labelled *constructor edge* $\mathsf{cons}^i$ from the sub-element to the node; and add a labelled *decomposition edge* $\overline{\mathsf{cons}}^i$ in the reverse direction. For example, $\delta_0$ 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 $\Rightarrow^1$ and $\Rightarrow^2$ respectively.

Repeat step 2 until no more edges or nodes are added. Figure 7 describes this process more formally. Most rules are straightforward, but two 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 constraint analysis (§5.3).

$$
\begin{array}{rl}
n: & \textbf{Node} \qquad\qquad\qquad\qquad (\textbf{Node} = \textbf{Element}) \\
e: & \textbf{Edge} \ ::= \mathsf{LEQ}\{H\}(n_1 \mapsto n_2) \\
& \quad \mid \ \mathit{con}^i(n_1 \mapsto n_2) \ \mid \ \overline{\mathit{con}}^i(n_1 \mapsto n_2)
\end{array}
$$

$$
\begin{array}{c}
\textbf{Graph} = (\wp(\textbf{Node}), \wp(\textbf{Edge})) \\
\mathcal{E}[\![E]\!] : \textbf{Graph} \qquad \mathcal{A}[\![G]\!] : \textbf{Graph}
\end{array}
$$

---

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

Figure 7: Construction of the constraint graph.

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: $(\mathit{fun}\ A = \mathit{fun}\ B) \not\Rightarrow A = B$.

## 5.2 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 discuss the challenge of analyzing Haskell constraints, and then propose a new algorithm that tackles these challenges.

**Limitations of previous approach** Graph saturation can be formalized as a *context-free-language (CFL) reachability* problem [24, 3, 36]. For example, Zhang and Myers formalized a graph saturation algorithm for a subset of our constraint language as the first three rules in Figure 8. The first rule infers a new LEQ edge given two consecutive LEQ edges, reflecting the transitivity of $\leq$. This rule also aggregates hypotheses made on existing edges to the newly inferred edge. The second rule infers a new LEQ edge when a constructor edge is connected to its dual decomposition edge, reflecting the fact that constructors can be decomposed. Given $n_i \leq n_i'$ for parameters of $\overline{n}$ and $\overline{n'}$, the third rule infers an LEQ edge from $\mathit{con}(\overline{n})$ to $\mathit{con}(\overline{n'})$, reflecting the fact that constructors are covariant.

However, graph saturation is insufficient to handle SCL. We can see this by considering the constraint graph of the running example, in Figure 6. Excluding the white nodes and the edges leading to and from them, this graph is fully saturated according to the rules in Figure 8. For example, the dotted edges between $\delta_0$ 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 6. Since this inequality reveals an error in the program being analyzed (that $[\mathsf{Char}]$ is not an instance of class $\mathsf{Num}$), failure to identify it means an error is missed. Moreover, the edges between $(\xi_2, \xi_2)$ and $(F\ \mathsf{a}_0)$ are mistakenly judged as unsatisfiable, as we explain in §5.3.

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

where $\mathit{con} \in \mathbf{Con}$, $\mathit{fun} \in \mathbf{Fun}$. First two rules apply for consecutive edges.

Figure 8: Graph saturation rules. New edges (left) are inferred based on existing edges (right).

**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 8

**Procedure** `expand`($G$ : ***Graph***, $\mathcal{T}$ : ***Element*** $\rightarrow$ ***Trace***)
  For a matched pattern shown in Figure 10, say $E_{\texttt{old}}$ is in $G$ already. Add $E_{\texttt{new}}$ to $G$. Let $E \le E'$ 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 $(\mathcal{T}(E_{\texttt{old}}),(E \hookrightarrow E'))$)

Figure 9: Graph saturation and expansion algorithm.

**Expanding the graph** The key insight for making the algorithm more sound and complete is to *expand* the constraint graph during graph saturation. Informally, nodes are added to the constraint graph so that the third and fourth rules in Figure 8 can be applied.

The (full) constraint graph in Figure 6 is part of the final constraint graph after running our new algorithm. The algorithm expands the original constraint graph with a new node $[\mathsf{Char}]$. Then, the dotted edge from $[\mathsf{Char}]$ to $[\alpha_0]$ is added by the third production in Figure 8, and then the dotted edge from $[\mathsf{Char}]$ to $\mathsf{Num}$ by the first production. Therefore, the unsatisfiable inequality $([\mathsf{Char}] \le \mathsf{Num})$ is correctly identified by the new algorithm. Moreover, the same mechanism identifies that $(F\ \mathsf{a}_0) = (\mathsf{b}_0, \mathsf{b}_0)$ can be entailed from hypothesis $H'$, as we explain in §5.3. 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 `[`$\alpha_0$`]`, `[Char]`, `[[`$\alpha_0$`]]`, `[[Char]]` and so on infinitely.

**The new 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 rules in Figure 7); *white nodes* are added during graph expansion.

The algorithm is shown in Figure 9. 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 8. The (new) fourth rule is needed for function applications, reflecting an axiom $\forall \overline{E_1}, \overline{E_2}\ .\ \overline{E_1} = \overline{E_2} \Rightarrow \mathit{fun}\ \overline{E_1} = \mathit{fun}\ \overline{E_2}$. We omit the details of graph saturation in this paper since it is mostly standard [3, 36].

The most interesting part is the procedure `expand`, which actively adds (only) new nodes to the graph, so the saturation procedure may saturate the graph further. As depicted in Figure 10, this procedure looks for an LEQ edge between some elements $E$ and $E'$ in the graph $G$. If $G$ contains only one of $\mathit{con}\ (E_1, ., E, ., E_n)$

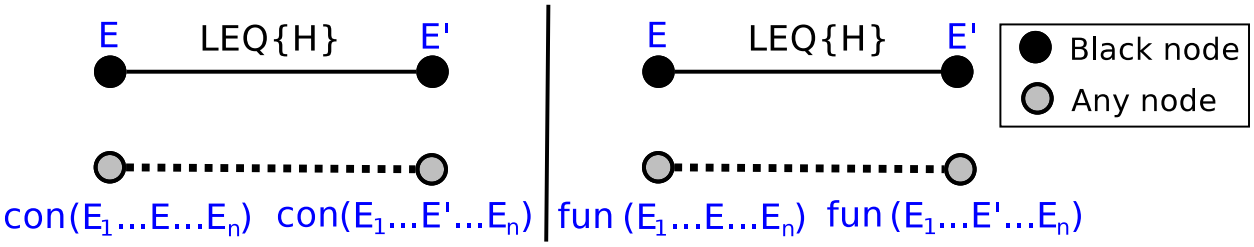



Figure 10: Graph-expanding patterns. If only one grey node is in the graph, the other one is added as a white node.

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 dotted edges in Figure 10).

To ensure termination, the expansion procedure puts a couple of restrictions on 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 sequence of a single black node, and 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 $\mathcal{T}(E_{\texttt{new}})$ (line 2).

Returning to our running example in Figure 6, the LEQ edge from $\alpha_0$ to Char, as well as the node $[\alpha_0]$, match the pattern in Figure 10. 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, $\mathcal{T}([\mathsf{Char}]) = ([\alpha_0], (\alpha_0 \hookrightarrow \mathsf{Char}))$.

**Termination** The algorithm in Figure 9 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).

### 5.3 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 6, the LEQ edge from $(\mathsf{b}_0, \mathsf{b}_0)$ to $(F\ \mathsf{a}_0)$ corresponds to the following entailment:

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

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

**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 final graph.

The construction of a hypothesis graph is shown in Figure 11. 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:

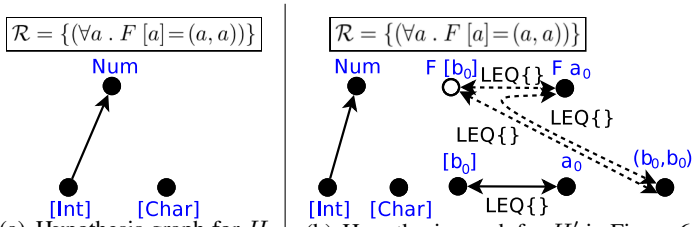
$\mathbf{HGraph} = (\mathbf{Graph}, \mathcal{R})$ $\quad \mathcal{R} = \wp(Q)$ $\qquad\qquad$ $\mathcal{Q}[\![H]\!] : \mathbf{HGraph}$

$\mathcal{N}$ : the constraint graph w/o edges $\qquad\qquad$ $\mathcal{A}[\![G]\!]$ : as defined in Figure 7

$$\mathcal{Q}[\![Q_1 \wedge \ldots \wedge Q_n]\!] = (\mathcal{N}, \emptyset) \cup \bigcup_{i \in 1..n} \mathcal{Q}[\![Q_i]\!]$$

$$\mathcal{Q}[\![I]\!] = (\mathcal{A}[\![\emptyset \vdash I]\!], \emptyset) \qquad \mathcal{Q}[\![\forall \overline{a} \,.\, C \Rightarrow I]\!] = (\emptyset, \{\forall \overline{a} \,.\, C \Rightarrow I\})$$

Figure 11: Construction of the hypothesis graph.




(a) Hypothesis graph for $H$ in Figure 6.

(b) Hypothesis graph for $H'$ in Figure 6.

Figure 12: Hypothesis graphs for the running example.

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. Consequently 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 12 (excluding the white node and dotted edges) shows (part of) the constructed hypothesis graphs for hypotheses $H$ and $H'$.

The hypothesis graph is then expanded and saturated in a similar way as the constraint graph. The difference is that axioms are applied during saturation, as shown in Figure 13. 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 12(b). The node $F\ [\mathsf{b}_0]$ is added by `expand` in Figure 9. Moreover, the quantified axiom $(\forall a \,.\, F\ [a] = (a, a))$ is applied, under the substitution $(a \mapsto \mathsf{b}_0)$. Hence, the algorithm adds the dotted edges between $F\ [\mathsf{b}_0]$ and $(\mathsf{b}_0, \mathsf{b}_0)$ to the hypothesis graph. The final saturated hypothesis graph contains edges between $F\ \mathsf{a}_0$ and $(\mathsf{b}_0, \mathsf{b}_0)$ 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 $(F\ \mathsf{a}_0)$ in Figure 6 are mistakenly classified as unsatisfiable.

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

1 Add new edges to $G$ according to the rules in Figure 8
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** $\exists \theta : \overline{a} \mapsto \overline{\textbf{\textit{Node}}} \,.\, \forall 1 \leq i \leq n \,.\, \theta[I_i] \in G$ **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

Figure 13: Hypothesis graph saturation for axioms.

**Classification** An entailment $H \vdash E_1 \leq E_2$ is classified as satisfiable iff there is a level-respecting substitution $\theta$ 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 example in Figure 6, the LEQ edges between $(\mathsf{b}_0, \mathsf{b}_0)$ and $F\ \mathsf{a}_0$ are (correctly) classified as satisfiable, since the corresponding edges are in Figure 12(b). LEQ edges between $(\xi_2, \xi_2)$ and $F\ \mathsf{a}_0$ are (correctly) classified as satisfiable as well, with substitution $\xi_2 \mapsto \mathsf{b}_0$. On the other hand, the LEQ edge from $[\mathsf{Char}]$ to $\mathsf{Num}$ is (correctly) judged as unsatisfiable, since the inequality is not present in the (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 $\emptyset \vdash (\beta_2 \rightarrow (\alpha_0, \beta_2)) = (\mathsf{a}_1 \rightarrow (\mathsf{a}_1, \mathsf{a}_1))$, which requires that the inferred type for the implementation of $\mathsf{g}$ be equivalent to its signature. The final constraint graph for the assertion contains two LEQ edges between nodes $\beta_2$ and $\mathsf{a}_1$. These edges are correctly classified as unsatisfiable, since the only substitution candidate, $\beta_2 \mapsto \mathsf{a}_1$, is not level-respecting.

If the signature of $\mathsf{g}$ is $(\forall a\ .\ a = \mathsf{Int} \Rightarrow a \rightarrow (a, a))$, the program is well-typed, since the parameter of $\mathsf{g}$ must be $\mathsf{Int}$. This programs 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 $\beta_2 \mapsto \mathsf{Int}$.

**Informative edges** When either end node of a satisfiable LEQ edge is an unification variable, its satisfiability is trivial and hence not informative for error diagnosis. Also uninformative is an LEQ edge derived from unsatisfiable edges. Only the informative edges are used for error diagnosis.

# 6 Bayesian model for ranking explanations

When unsatisfiable edges are detected, we are interested in inferring the program expressions that (generated the constraints that) *most likely* caused the errors. To do this, we extend the Bayesian model of [36].

The observed symptom of errors is a fully analyzed constraint graph (§5), in which all informative LEQ edges are classified as satisfiable or unsatisfiable. For simplicity, in what follows we write "edge" to mean "informative edge".

Formally, an observation $o$ is a set $(o_1, o_2, \ldots, o_n)$, where $o_i \in \{\mathtt{unsat}, \mathtt{sat}\}$ represents satisfiability of the $i$-th edge. Let $\mathcal{E}$ be the set of all expressions in a program, each occurring in a distinct source location and giving rise to a typing constraint. We are looking for a set $E \subseteq \mathcal{E}$ that maximizes $P(E|o)$, the posterior probability that the expressions $E$ contain errors. By Bayes' theorem, this term has an easier, equivalent form: $P_{\mathcal{E}}(E) \times P(o|E)/P_{\mathcal{O}}(o)$, where $P_{\mathcal{E}}(E)$ is a prior probability that expressions in $E$ contain errors, and $P_{\mathcal{O}}$ is a prior distribution on observations. Since $P_{\mathcal{O}}(o)$ does not vary in $E$, the goal of error diagnosis is to find:

$$\arg\max_{E \subseteq \mathcal{E}} P_{\mathcal{E}}(E) \times P(o|E)$$

**Redundant edges** To further simplify the term $P_{\mathcal{E}}(E) \times P(o|E)$, [36] assume that the satisfiability of informative edges is independent. However, the introduction of white nodes undermines this assumption. In Figure 6, the satisfiability of the edge between $[\alpha_0]$ and $[\mathsf{Char}]$ merely repeats the edge between $\alpha_0$ and $\mathsf{Char}$; 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 1** *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 2 (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 $\tau_i$ and $\tau_i'$, 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 $\tau_i$ and $\tau_i'$, 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, $\tau_{0i}$ is a black node in $G$. Hence, the algorithm also adds $\tau_i$ by applying the same substitutions on $\tau_{0i}$, as well as $\tau_i'$ 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 3 (Path simulation 1)** *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 $\tau_i$ and $\tau_i'$, either $\tau_i = \tau_i'$, or there is a path from $\tau_i$ to $\tau_i'$ in $G$.*

**Proof**. We prove by induction on the path length. The base case (length=1) is trivial by Lemma 2.

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: $\mathsf{con}\ \overline{\tau_1} - \mathsf{con}\ \overline{\tau'} - \mathsf{con}\ \overline{\tau_2}$ for some $\tau'$. Result is true by Lemma 2 and induction hypothesis unless both $\mathsf{con}\ \overline{\tau'}$ and $\mathsf{con}\ \overline{\tau_2}$ are black nodes.

When both $\mathsf{con}\ \overline{\tau'}$ 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 $(\tau_i', \tau_{i2})$ by the second production in Figure 8. By Lemma 2, there is an edge connecting $\tau_{i1}$ and $\tau_i'$. Therefore, there is a path from $\tau_{i1}$ to $\tau_{i2}$ if they are different. □

**Lemma 4 (Path simulation 2)** *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 $\tau_i$ and $\tau_i'$, either $\tau_i = \tau_i'$, or there is a path from $\tau_i$ to $\tau_i'$ in $G$.*

**Proof**. We prove by induction on the path length. The base case (length=1) is trivial by Lemma 2.

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

By assumption, the first 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 $\tau'$. Result is true by Lemma 2 and induction hypothesis. □

**Proof of Lemma 1** 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 $\tau_{1i}$ and $\tau_{2i}$, either $\tau_{1i} = \tau_{2i}$ or there is a path from $\tau_{1i}$ to $\tau_{2i}$ 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 4 instead of Lemma 3 in the proof.

□

**Calculating likelihood** Let $\widehat{o} = (\widehat{o}_1, \widehat{o}_2, \ldots, \widehat{o}_m)$ be all non-redundant edges. Lemma 1 implies that $P_{\mathcal{E}}(E) \times P(o|E) = P_{\mathcal{E}}(E) \times P(\widehat{o}|E)$. We make two simplifying assumptions:

1. All expressions are equally likely to be wrong (with fixed probability $P_1$), and
2. Remaining paths in $\widehat{o}$ are independent.[2]

These assumptions allow us to rewrite $P_{\mathcal{E}}(E) \times P(\widehat{o}|E)$ as $P_1^{|E|} \times \prod_i P(\widehat{o}_i|E)$. The term $P(\widehat{o}_i|E)$ is calculated using two heuristics:

1. If $\widehat{o}_i = \texttt{unsat}$, at least one constraint that gives rise to the edge must be wrong. Therefore, we only need to consider the expressions that generate constraints along unsatisfiable edges in $\widehat{o}$. We denote this set by $\mathcal{G}$.
2. If $\widehat{o}_i = \texttt{sat}$, it is unlikely (with fixed probability $P_2 < 0.5$) that expressions in $E$ give rise to the edge.

Assume that constraints generated for $E$ appear on $k_E$ of satisfiable edges. Using the previous heuristics, the likelihood is maximized at:

$$\arg\max_{E \subseteq \mathcal{E}} P_1^{|E|} \prod_i P(\widehat{o}_i|E) = \arg\max_{E \in \mathcal{G}} P_1^{|E|} (P_2/(1-P_2))^{k_E}$$

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). We use the efficient A* search algorithm in [36] to find a set of expressions minimizing this metric.

[2]These assumptions might be refined to improve accuracy. For example, the (rare) missed locations in our evaluation usually occur because programmers are more likely to misuse certain operators (e.g., ++ and :) than others in Haskell. We leave refining these assumptions as future work.

# 7 Implementation

We built our error diagnostic tool based on the open-source tool SHErrLoc [30]. Our diagnostic tool reads in constraints following the syntax of Figure 2, and computes constraints most likely to have caused errors in the constraint system being analyzed. The extension includes about 2,500 lines of code (LOC), above the 5,000 LOC of SHErrLoc.

Generating constraints from Haskell type inference involved little effort. 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 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.

# 8 Evaluation

## 8.1 Benchmarks

To evaluate our error diagnosis tool, we used two sets of previously collected Haskell programs containing errors. The first corpus (the CE benchmark) contains 121 Haskell programs, collected by [6] 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 [11], contains over 50k Haskell programs logged by Helium [13], 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.

## 8.2 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 [15]; 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.

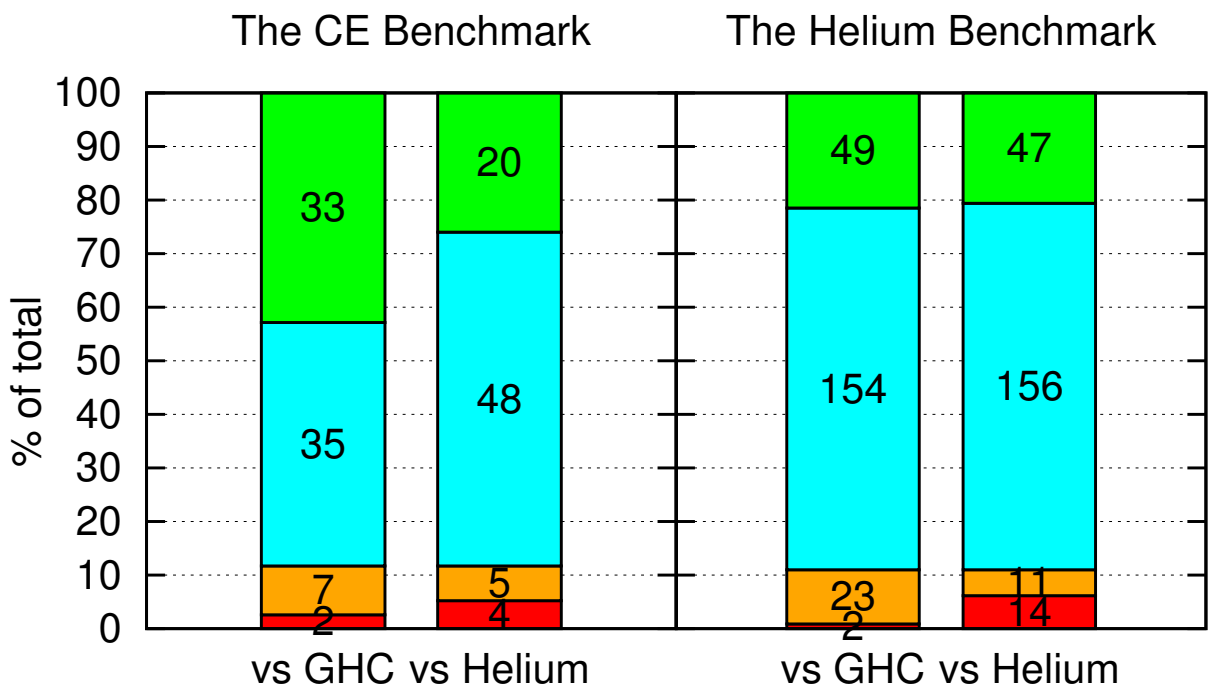



Figure 14: Comparison with GHC and Helium on two benchmarks. From top to bottom, columns count programs where (1) our tool finds a correct error location that the other tool misses; (2) both tools report the correct error location; (3) both approaches miss the error location; (4) our tool misses the error location but the other tool finds one of them.

### 8.3 Error report accuracy

Figure 14 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 finds the true error locations for 48%, 68% and 88% of programs. 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%).

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** [6] 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

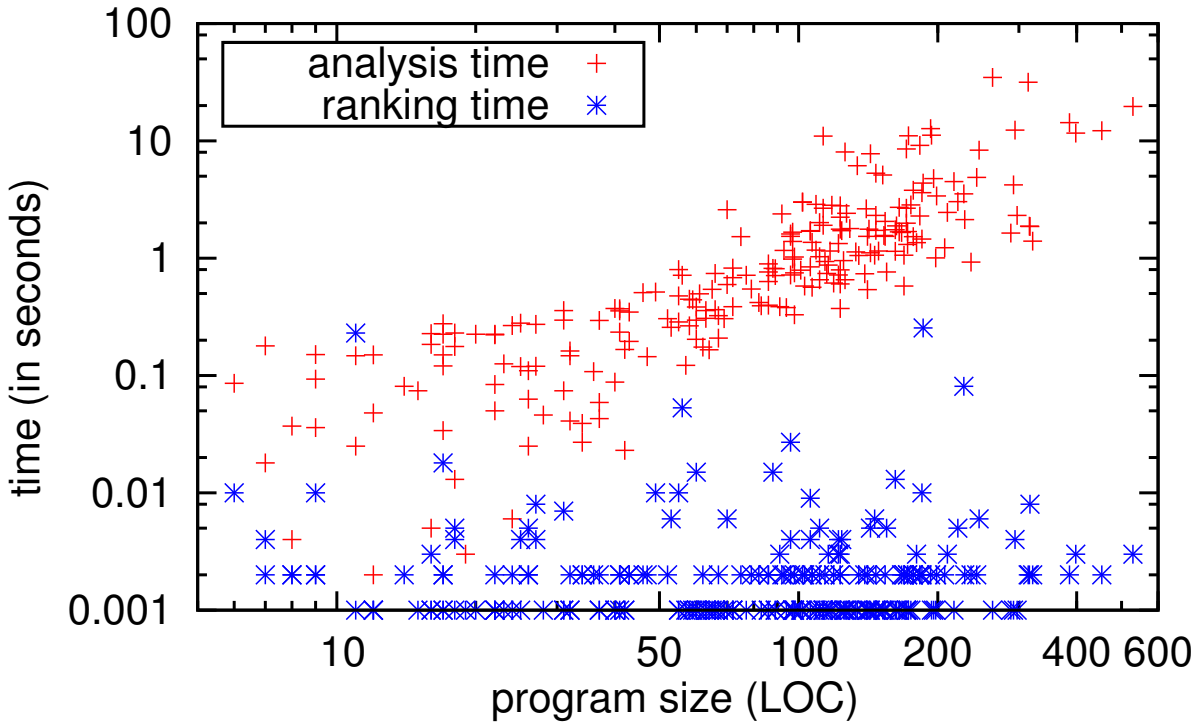



Figure 15: Performance on the Helium benchmark.

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[3]. When our tool reports suboptimal suggestions, the accuracy becomes 92% , with an average suggestion size of 3.2.

**Comparison with SHErrLoc** Zhang and Myers evaluated their error diagnosis algorithm using a suite of OCaml programs collected from students by [20]. We checked that our extensions to their SHErrLoc tool did not harm accuracy. Using their benchmark data, in which true errors are already labeled, and their constraint generation process, we found that accuracy is unaffected by our extensions. This result is expected since OCaml programs use none of the advanced features that this paper targets.

### 8.4 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 (§5) and by ranking (§6).

**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 15 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

[3] 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 [4].

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 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.

### 8.5 Sensitivity

Recall (§6) 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). 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 rely on carefully choosing the ratio.

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 $\sim$1.7). Hence, satisfiable paths *are* important for error diagnosis.

## 9 Related work

The most closely related work is clearly that of [36]. In order to handle the highly expressive type system of Haskell, it was necessary to significantly extend many aspects of that work: the constraint language and constraint graph construction, the graph saturation algorithm, and the Bayesian model used for ranking errors.

**Error diagnoses for ML-like languages** Efforts on improving error messages for ML-like languages can be traced to the 80's [35, 16]. Most of these efforts can be categorized into three directions.

The first direction, followed by [16, 19, 22, 14, 36, 26] as well as most ML-like language compilers, attempts to infer the *most likely* cause. One approach is to alter the order of type unification [19, 22, 5]. But any specific order fails in some circumstance, since the error location may be used anywhere during the unification procedure. Some prior work [16, 14, 12, 36, 26] 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.

The second direction [35, 7, 32, 31, 10, 9, 29], attempts to trace everything that contributes to the error. Despite the attractiveness of feeding a full explanation to the programmer, the reports are usually verbose and hard to follow [14].

A third approach is to fix errors by searching for similar programs [23, 20] or type substitutions [6] that do type-check. Unfortunately, we cannot obtain a common corpus to perform direct comparison with [23]. On the suite of OCaml programs used in [36], our tool improves on accuracy of [20] by 10%. The results on the CE benchmark (§8.3) suggests that our tool localizes true error locations more accurately than in [6]. Although our tool currently does not provide suggested fixes, accurate error localization is likely to provide good places to search for fixes.

**Constraints and graph representations for type inference** Modeling type inference via constraint solving is not a new idea. The most related work is on set constraints [1, 2] and type qualifiers [9]. Like SCL, this work has a natural graph representations, with constraint solving strongly connected to CFL-reachability [24, 17]. However, neither set constraints nor type qualifiers handle the hypotheses and type-level functions essential to representing Haskell constraints.

**Probabilistic inference** More broadly, other work in the past decade has explored various approaches for applying probabilistic inference to program analysis and bug finding. This work is summarized by [36].

# 10 Conclusion

We have shown how to use probabilistic inference to effectively localize errors for the highly expressive type system of Haskell. This contribution is clearly useful for Haskell programmers. However, because Haskell is so expressive, success with Haskell suggests that the approach has broad applicability to other type systems.

# 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*, pages 31–41, 1993.

[3] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30:809–837, 2000.

[4] S. Chen. Accuracy of CF-typing. Private communication, 2014.

[5] S. Chen and M. Erwig. Better type-error messages through lazy typing. Technical report, Oregon State University, 2014.

[6] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL 41*, Jan. 2014.

[7] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical report, Indiana University, December 1995.

[8] EasyOCaml. `http://easyocaml.forge.ocamlcore.org`.

[9] 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.

[10] 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.

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

[12] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. 2007.

[13] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pages 62–71, 2003.

[14] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.

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

[16] G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL 13*, pages 44–57, 1986.

[17] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. In *PLDI'04*, PLDI '04, pages 207–218, 2004.

[18] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, July 1998.

[19] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Prog. Lang. Syst.*, 20(4):707–723, 1998.

[20] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI'07*, pages 425–434, 2007.

[21] S. Marlow and S. Peyton-Jones. The Glasgow Haskell compiler. `http://www.aosabook.org/en/ghc.html`, 1993.

[22] B. J. McAdam. On the unification of substitutions in type inference. In *Implementation of Functional Languages*, pages 139–154, 1998.

[23] B. J. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, 2001.

[24] 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.

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

[26] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *OOPSLA'14*, pages 525–542, 2014.

[27] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, Jan. 2007.

[28] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced topics in types and programming languages*, pages 389–489. MIT Press, 2005.

[29] 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.

[30] SHErrLoc (Static Holistic Error Locator) tool release (ver 1.0). `http://www.cs.cornell.edu/projects/sherrloc`, 2014.

[31] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 72–83, 2003.

[32] 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.

[33] D. Vytiniotis, S. P. Jones, T. Schrijvers, and M. Sulzmann. Outsidein(X): Modular type inference with local assumptions. *Journal of Functional Programming*, September 2011.

[34] D. Vytiniotis, S. Peyton Jones, and T. Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 39–50, New York, NY, USA, 2010. ACM.

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

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