Goanna: A Novel Approach for Automated Type Error Debugging

Shuai Fu, Tim Dwyer, Peter J. Stuckey, John Grundy

Monash University.

Contributing authors: shuai.fu@monash.edu; tim.dwyer@monash.edu; peter.stuckey@monash.edu; john.grundy@monash.edu;

Abstract

Statically typed languages offer many advantages in software engineering, including bug prevention, enhanced code quality, and reduced maintenance costs. However, these benefits come at the expense of a steep learning curve and a slower development pace. While known for its expressive and strong type system, Haskell often frustrates programmers when learning and using its powerful type system, especially when debugging type errors. We introduce Goanna, a novel tool that performs type-checking for Haskell and focuses on improving error diagnostics. When encountering type errors, Goanna identifies a comprehensive list of possible causes and provides a potential course of action for each cause. To achieve this, Goanna uses constraint logic programming and Minimal Correction Subsets (MCS) to support its diagnostics. We evaluated Goanna's effectiveness using 153 diverse Haskell programs from online discourse and research literature, demonstrating its ability to accurately and reliably identify root causes compared to traditional tools. Our benchmark study shows that Goanna performed modestly when diagnosing large programs containing complex errors, but was responsive in providing real-time debugging assistance when working with small to medium-sized programs.

Keywords: Type Errors, Constraint Logic Programming, Functional Languages, Integrated Development Environment

1 Introduction

Statically typed languages offer numerous benefits in software engineering, including the ability to detect errors before program execution [1, 2], facilitating easier and

safer code refactoring [3], enhancing code readability [4], and providing tighter integration with code editors and integrated development environments (IDEs) [5]. These attributes often render statically typed languages more suitable for large projects that undergo frequent modifications. The emergence of new languages with strong type systems, such as Rust and TypeScript, and the addition of static type checkers to previously dynamically typed languages, like MyPy for Python and static types for PHP and Ruby, reflect this trend. In recent years, the growing use of large language models for code generation renewed the demand for statically typed languages, as they detect certain errors before running the program, providing confidence in security and code quality that generated code often lacks [6–9].

Statically typed languages manifest in multiple paradigms. Functional programming languages, such as ML and Haskell, have embraced static typing since their conception. Compared to languages from other paradigms, such as Java and C, statically typed functional languages offer two distinct advantages. First, they feature more flexible and expressive type systems that allow domain problems to be succinctly modeled at the type level with strong guarantees of correct behavior. Second, the support for polymorphic types and global type inference enables rigorous type checking even without explicitly writing type annotations. This paper uses Haskell as a case study to explore the challenges and solutions associated with advanced type systems.

Haskell is renowned for its expressive and robust type system, which allows programmers to develop programs in a type-driven manner. Innovations in type systems frequently pioneered from Haskell, such as algebraic data types, type inference, and type classes, have been incorporated into mainstream programming languages [10–13]. Despite the advanced type system, Haskell is equally known for its steep learning curve, unforgiving type checker, and cryptic type errors. Extensive research has sought to alleviate these challenges [14–19]. Programmers often find type error messages in Haskell are excessively terse, confusing, and misleading. For example, as illustrated in Fig. 1, a type mismatch between a `Char` and an integer leads to perplexing errors, particularly for novice users. We have identified three main challenges to resolve the underlying type error using the existing error reporting:

- **Challenge 1:** The errors are ***biased towards a single potential cause***, neglecting other plausible causes;
- **Challenge 2:** Making changes to the suggested location ***does not always resolve the error entirely***; and
- **Challenge 3:** ***Insufficient contextual information is provided***, hindering the programmer's ability to understand the underlying rationale of the error message.

In this paper, we first present a method that categorizes type errors into four basic forms based on constraint logic. This categorization helps understand and reason about complex errors in practice and informs the way we discuss and diagnose type errors. To tackle the challenges of diagnosing and resolving type errors in statically typed languages like Haskell, we introduce a novel tool: *Goanna* [1]. Goanna is a standalone Haskell type checker designed to address the aforementioned challenges. Unlike traditional type-checking tools, which report a single possible cause of the type

[1] Goanna is available at https://goanna.typecheck.me

```
students = [(1, 99), (2, 60), (3, 55)]
matchFirst key (k, v) = k == key
filterById =
  filter (matchFirst '1') students

scores = [99, 60, "55"]
```

```
ghc ./Main.hs
Main.hs:1:14: error:
    • No instance for (Num Char) arising
from the literal '1'
    • In the expression: 1
      In the expression: (1, 99)
      In the expression: [(1, 99), (2,
60), (3, 55)]
  |
1 | students = [(1, 99), (2, 60), (3, 55)]
  |
```

Fig. 1 Inspecting a type error using the Haskell compiler GHC (Glasgow Haskell Compiler)

error, Goanna finds all potential causes (Challenge 1); Goanna suggests accurate and reliable locations, which, when modified, will fully resolve the type error(Challenge 2); Goanna provides contextual type hints to support easier resolution (Challenge 3). Additionally, we present Goanna-IDE (Section 4), an interactive programming environment for Haskell that facilitates visualization and easy navigation of Goanna's type error diagnostics. Goanna and GoannaIDE are open-source, and an online demo is publicly available [20].

We conducted a corpus study on two datasets to assess Goanna's accuracy and reliability (Section 6.2), while our benchmark study evaluates Goanna's performance (Section 6.3). Results demonstrate that Goanna consistently provides accurate and reliable error diagnostics compared to conventional tools. Goanna shows performance constraints when diagnosing large programs containing complex type errors. However, Goanna is able to provide real-time debugging assistance when working with small to medium programs.

The main contributions of this research are:

- A categorization of type errors based on constraint logic that helps to reason about complex type errorsn(Section 3).
- Goanna [20], a Haskell type checker with advanced error reporting capabilities.
- Goanna-IDE, an interactive Haskell development environment that supports debugging type errors using Goanna's diagnostics. (Section 4)
- A rigorous evaluation of Goanna's accuracy, reliability, and performance. (Section 6)

The techniques employed in Goanna, such as enumerating all possible causes and ranking potential causes based on their likelihood to be the root cause, are not confined to the Haskell language. Rather, they are broadly applicable to statically typed programming languages, and Goanna is designed for adaptability across languages with similar type systems.

2 Related Work

This section begins with a review of constraint-based type systems, which is the theoretical framework that informs Goanna's development. We list notable work in constraint-based type systems and analyze their techniques. We then explore studies with similar objectives to Goanna but employing alternative approaches. Finally, we review research focused on interactive debugging of type errors and discuss its relevance to GoannaIDE.

2.1 Constraint-based Type Systems

Type systems can be viewed as logical systems, a perspective established since the dawn of programming language research [21].

Constraint logic, a branch of formal logic, offers an intuitive and declarative framework for understanding types as logic constraints. In constraint logic, type checking a program can be viewed as finding solutions for possible type assignments for each expression in the source code. Each expression introduces new restrictions on potential solutions. Many studies [22, 23] define type systems based on pure constraints. To gain more insights from an ill-typed program (in other words, an infeasible constraint set), a commonly used technique is to find and analyze a few critical subsets, defined as follows:

Minimal Unsatisfiable Subset (MUS) An MUS of a set of constraints C is a subset $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable. An MUS provides a concise explanation for the infeasibility of a system, highlighting a logical pathway from one conflicting location to another. Thereby, MUS is often associated with addressing the "Why" question [24, 25], in other words, finding the minimal contributors that prevent the system from emitting a defect.
Minimal correction set (MCS) An MCS of a set of constraints C is a subset $M \subseteq C$ such that $C \setminus M$ is satisfiable and $\forall S \subset M : C \setminus S$ is unsatisfiable. MCSes are so named because their removal from C can be seen to "correct" the infeasibility. Thus, MCS is often associated with addressing the "Why not" question [24, 25], in other words, finding the minimal reasons that trigger a certain defect.
Maximal satisfiable subset (MSS) An MSS of a set of constraints C is a subset $M \subseteq C$ such that M is satisfiable and $\forall c \; in \; C \setminus M : M \cup \{c\}$ is unsatisfiable. The definition of an MSS is symmetric to that of a MUS, with "satisfiable" and "unsatisfiable" swapped, along with "maximal" for "minimal". MCS and MSS are the complement sets of one another. In practical terms, an MSS represents the largest portion of the source program that does not contribute to the type error. It can be used to provide a best-effort type assignment by ignoring the smallest section that caused the type error.

Haack and Wells' research on type error slicing [26] presents a method for identifying all locations related to a type error, providing improved error reporting for programmers seeking to understand the underlying causes. The system introduced employs the concept of MUS, though it uses a different terminology (Minimal Unsolvable Constraint Sets), to identify the smallest set of locations that contribute to the error. Similarly, Chameleon [27] utilizes type error slicing as its foundational method,

developed on a generalized and efficient constraint semantics (constraint handling rules [28]). Chameleon supports advanced features such as type classes and functional dependent types. Pavlinovic, King, and Wies introduced a constraint-based type system using weighted maximum satisfiability modulo theories (MaxSMT) along with a state-of-the-art MUS identification algorithm (QuickExplain) [29]. The above approaches depend on a single MUS to determine relevant error locations. In contrast, Goanna distinguishes itself by identifying all MUSes, MSSes, and MCSes. This allows Goanna to perform more comprehensive analyses, potentially leading to more accurate and informative diagnostics. For instance, none of the above work shares Goanna's ability to identify all possible causes of a type error.

The process of finding all MUSes, MSSes, and MCSes is often referred to as *ALLMUS*. *ALLMUS* has been studied by many [30–35]. The difference in technique of these studies is beyond the scope of this paper. Although many studies on type error improvement use a single MUS, Goanna is the first tool to apply *ALLMUS* in this field.

2.2 Finding all possible causes of type error

One key feature of Goanna is its ability to identify multiple possible causes for type errors based on MUS enumeration. However, other approaches can achieve similar outcomes. SHErrLoc [17] provides a type system modeled in a custom constraint language (SCL). SHErrLoc employs advanced graph analysis techniques for type inference and diagnosis. It can accurately identify type errors and propose all possible causes for Haskell and ML. Both Goanna and SHErrLoc employ constraint-based type inference; however, Goanna uses a standardized constraint language (Prolog) and employs well-established analysis techniques, such as MUS and MCS. This allows Goanna to potentially find easier paths to optimize its capabilities and performance, thanks to the maturity of these techniques.

Counter-factual Typing (CFT) [15, 36, 37] adopts a variation-based type system grounded in choice calculus [38]. Despite differing approaches, CFT and Goanna share key similarities, including their ability to provide multiple possible causes for type errors, offer the correct type for each cause, and employ ranking heuristics to estimate the likelihood of each cause. While CFT uses specialized type inference algorithms rooted in variation-based semantics, Goanna adopts a modular design, employing general-purpose tools and techniques, with only its constraint generation being specific to the language.

These methods face similar performance limitations; complex type error conditions can lead to an arbitrarily large number of potential causes, and in turn, significant computational demands. The authors of SHErrLoc recognize that its performance is suboptimal, and the testing was only performed on small Haskell programs containing fewer than 600 lines of code. Performance analysis was conducted in the work on Counter-Factual Typing [15], revealing that running times can become exponential with unbounded nesting levels of choices. The Efficient CFT [36] approach, which enhances performance through a brute-force optimization strategy, is based on the notion that changes in the program are incremental, allowing later type checks to benefit from earlier computational results. Goanna adopts a divide-and-conquer

strategy. We categorize type errors into a few atomic categories, evaluated Goanna's performance based on the categorization, and devised effective pathways for each category.

2.3 Interactive Type Error Debugging

A key contribution of our work is the interactive type debugging interface, GoannaIDE. While debugging tools specialized for type errors have not been extensively studied over the years, GoannaIDE draws inspiration from a few existing systems. One such system, ChameleonIDE [39], enhances traditional type error slicing in Chameleon [27] by enabling programmers to efficiently comprehend type errors through interactive exploration of the logical reasoning chain in the MUS. The primary distinction between ChameleonIDE and GoannaIDE lies in their focus; ChameleonIDE addresses the "why" behind type errors, emphasizing the explanation of inference steps, whereas GoannaIDE concentrates on the "how," focusing on investigating potential causes and identifying the minimal changes needed to achieve a resolution.

Additionally, systems such as Hazel [40] and Hazel Tutor [41] introduce innovative concepts like structural editing and type-hole driven development. Hazel emphasizes providing an editing experience that maintains *livelihood*, meaning the code remains syntactically valid and can be typed at all times. In contrast, GoannaIDE has a narrower focus: it offers tools designed to help programmers navigate from a type error to a well-typed program efficiently.

3 Categories of Type Error

Before introducing Goanna, we present a categorization of type errors based on the presence of minimal unsatisfiable subsets (MUS). All complex type errors encountered in practice can be decomposed into one or more of four basic categories. This formalization helps reason about the cause of type errors and provides intuitive explanations for them. We use this categorization as a guiding principle for designing Goanna's error diagnosis. Furthermore, this categorization offers insights into the computational complexity of our method, as demonstrated in Section 6.

3.1 Simple Type Error

A **simple type error** always involves two locations in the program that can not agree on a type assignment. An example like Listing 1 shows such an error. In the example, `x` can be inferred as either `Int` or `String` type based on line 1 or line 2.

Listing 1 An example of a simple type error

```
x :: Int
x = "3"
```

In practice, type errors in this category are trivial to resolve. The two offending locations are close to each other, and they do not require special mental bookkeeping to trace back to the root cause.

3.2 Multi-step Type Error

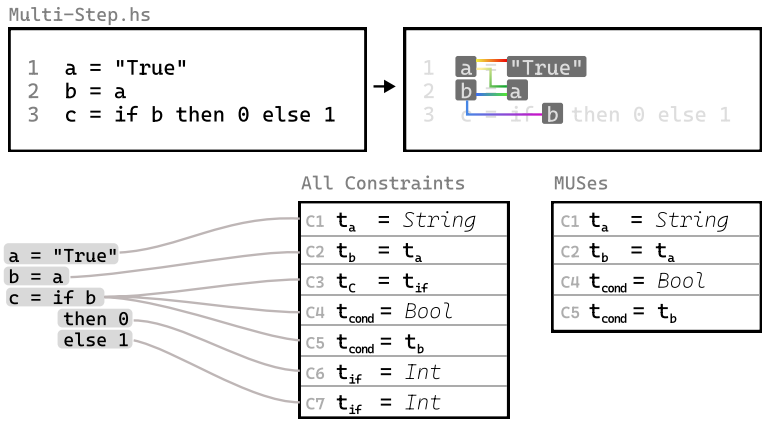

Fig. 2 A multi-step type error influenced by potential issues in the definition of `a`, `b`, or the conditional expression involving `b`. These elements are logically connected as depicted (Top Right). The analysis of the source code yields 7 constraints (Bottom Left), from which only one Minimal Unsatisfiable Subset (MUS) is identified (Bottom Right). In this, the notation t_x is used to denote a meta-variable. A meta-variable represents the potential type an expression is going to assume.

Multi-step type errors involve a sequence of logical deductions that link one conflicting location to another. When viewed as a constraint system, a multi-step type error is a type error that contains a single MUS. Relaxing any constraint within the MUS leads to a satisfiable system, implying that each location correlated to a constraint in the MUS could potentially cause the type error. For instance, in a typical multi-step type error shown in Fig. 2, the MUS consists of the constraints $\{C1, C2, C4, C5\}$. Removing any one of the elements would allow the program to type-check successfully.

For simplicity, in this paper, we only discuss one type of constraint: equality, written as $A = B$ to denote that A and B can be unified using the unification algorithm [42]. Other types of constraints are needed to specify advanced type-level features, such as type classes. In this, the notation t_x is used to denote a meta-variable. A meta-variable represents the potential type that an expression is going to assume.

Multi-step type errors are more common in practice. They occur naturally when programs develop layers of indirection or abstraction. They can cause confusion to programmers because the error can be reported in one place, but the root cause is at a distant location and requires tracing back and forth and mental bookkeeping.

3.3 Multi-witness Type Error

A multi-witness type error occurs when multiple locations (witnesses) on one side of a conflict suggest the same type assignment. When viewed as a constraint system, a multi-witness type error contains multiple MUSes. The precise number of MUSes depends on the number of witnesses at each endpoint. If a type error involves m witnesses on one side and n on the other, there will be a total of $m \times n$ MUSes.

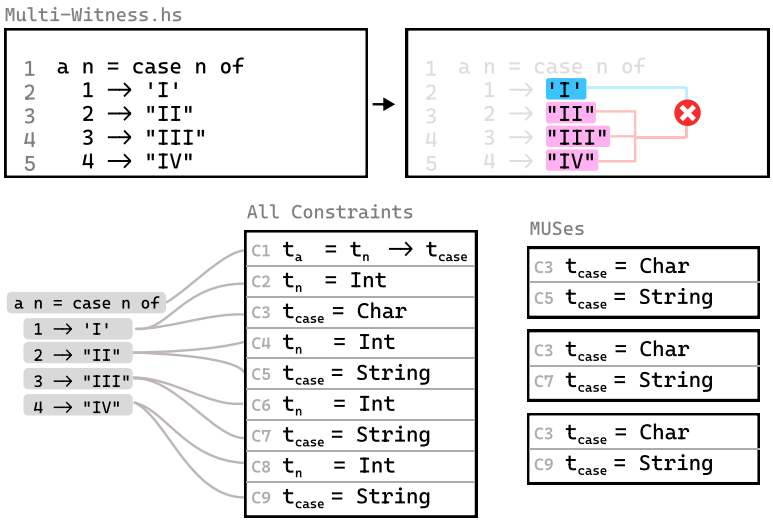

Fig. 3 Analysis of a multi-witness type error, where the `case` expression could be interpreted as type `Char` or `String`. This scenario showcases a type error with three witnesses against one witness (Top Right). From the source code, 9 constraints are established (Bottom Left), resulting in 3 identified MUSes. Interestingly, all MUSes include constraint C3, suggesting its pivotal role in favoring the possibility that the `Char` type is erroneous.

For example, for a multi-witness type error shown in Fig. 3, there are three MUSes: $\{C3, C5\}, \{C3, C7\}, \{C3, C9\}$. It is crucial to note that constraint $C3$ appears in every MUS, signaling a likely root cause.

Multi-witness type errors are also common in practice. They often occur when changes to the definition of one fragment of the program, and this fragment is referenced by multiple places. Like multi-step type errors, they too often require programmers to search many locations to arrive at the correct resolution.

3.4 Multi-party Type Error

A multi-party type error features several irreconcilable type assignments stemming from different locations within the source code. Like multi-witness errors, multi-party errors contain multiple MUSes. In the multi-party type error shown in Fig. 4, there are three MUSes: $\{C3, C4\}, \{C3, C5\}, \{C4, C5\}$. Unlike the multi-witness scenario, no single element appears across all MUSes, suggesting a more complex type error scenario. Generally, if a multi-party type error contains n parties, then there are $\binom{n}{2}$, or $n(n-1)/2$ unique MUSes.

Multi-party type errors are not common in real-life programs. They can happen when programmers mix the order of multiple arguments in a function application.

We provide strategies to improve type error diagnostics for each category:

1. **Simple Type Errors:** Provide a clear indication of the pair of locations associated with the error and explain what the logical relationship connects this pair.

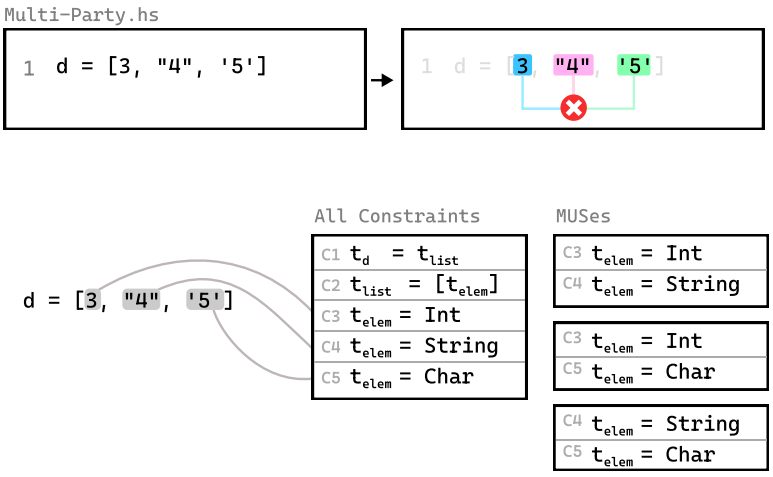

Fig. 4 Analysis of a multi-party type error regarding the list `d`, which could alternately be typed as `[Int]`, `[String]`, or `[Char]`. The conflict is depicted as a disagreement among three parties (Top Right). An analysis based on the source code generates 5 constraints (Bottom Left), leading to 3 distinct Minimal Unsatisfiable Subsets (MUSes). Notably, this scenario differs from the multiple witness type error as no single constraint appears across all MUSes.

2. **Multi-step Type Errors:** To support easier tracing of the propagation of type errors through the steps. Provide a clear indication of the pair of locations associated with each step and explain what the logical relationship connects with them.
3. **Multi-witness Type Errors:** Identify the witnesses and their associated locations on each side of the conflict. Allow programmers to quickly understand the discrepancy in the number of witnesses between the two sides.
4. **Multi-party Type Errors:** Breaking down these errors into simpler forms to support easier understanding and resolution. Allow the programmer to flexibly choose which subpart to focus on.

4 An Overview of Goanna

In this section, we demonstrate Goanna's features using some real-world examples. To showcase the debugging features, we developed Goanna-IDE, an interactive development environment for Haskell specializing in type error debugging. It makes extensive use of Goanna's type error diagnostics and delivers it to programmers through visualization and interactivity. An online demo of Goanna-IDE is available for evaluation at [43]. Goanna-IDE includes a file explorer, a text editor, and a debugging panel. Goanna-IDE provides the following key features when type errors are encountered:

- Thoroughly detect all type errors within the codebase and allow users to inspect each type error individually via the debugging panel.

- Indicate the most likely causes by star indicators.
- Show necessary type hints in the editor panel to help reason about each possible cause.

4.1 Examples of Diagnosing Type Errors with Goanna-IDE

For the type error in the motivating example, Goanna shows 6 possible causes of the error (see top right corner of Fig. 5). When focusing on the cause suggested by GHC, instead of highlighting only the literal `1`, Goanna reports all 3 literals that needed to be changed together if the programmer chooses to address this cause. In this typical multi-witness type error (Section 3.3), Goanna successfully identified all 3 witnesses and reported them all at once, a feature the traditional tool failed to perform in Fig. 1. In addition, Goanna also indicates that these integer literals need to be changed to `Char` type using the inlay type hints on line 1, largely narrowing down the potential ideal fixes.

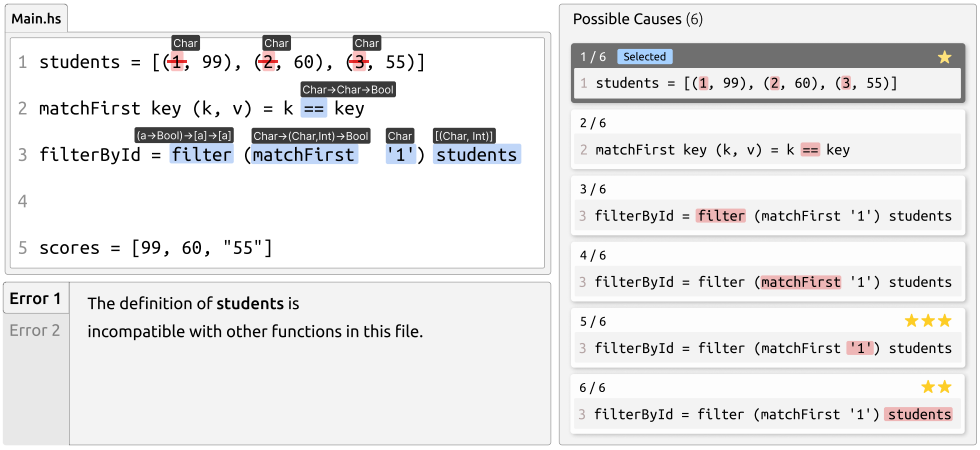

Fig. 5 Goanna's error diagnosis Goanna shows that to fix the type error, the literals `1`, `2`, and `3` on line 1 need to be changed to Char type.

Note that the cause suggested by GHC is only one of the possibilities identified by Goanna. In fact, Goanna suggests that there are more likely fixes, indicated by the star symbols. Goanna-IDE uses a star-based rating system to signal the "likelihood" of each cause. Three stars indicate the most likely cause, 2 stars, and 1 star follows. The most likely fix, based on Goanna's cause heuristics (Section 5.4), is the `Char` literal `'1'` on line 5, indicated by the 3 stars (Fig. 5).

By clicking on the most likely cause, Goanna shows different highlights in the editor (e.g., see Fig. 6). Goanna reports that the error is caused by the literal `'1'` and suggests changing it to an integer. All the type hints are adjusted based on our new assumption. Goanna ranks all possible causes using a series of heuristics. In this case,

the preference is largely influenced by how many locations are required to change to fix the error.

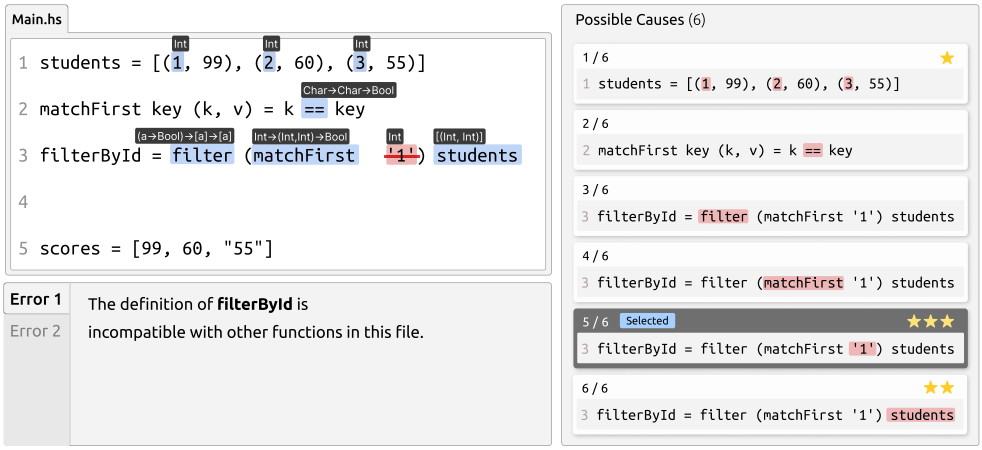

Fig. 6 Goanna's error diagnosis. Goanna shows that the type error can be fixed by changing the literal '1' on line 3, which needs an `Int` type. This, according to Goanna, is the most likely cause of the type error.

In fact, **the causes identified by Goanna are comprehensive**. Goanna will take into account potential causes in expressions, pattern matchings, type annotations, and type class constraints. Programmers will eventually find the root cause by exploring Goanna's diagnosis.

In addition, **each cause suggested by Goanna is sufficient to resolve the type error**. Traditional tools often reveal a set of partial locations of a type error, leaving programmers to realize later that additional adjustments are needed for a complete resolution. Goanna avoids this whack-a-mole-style error reporting by offering suggestions that encompass the complete set of changes necessary for a resolution.

4.2 Identifying all type errors

A key feature of Goanna is its ability to detect all type errors in the code, thanks to its MUS enumeration (Subsection 5.2). This is not always the case with other tools, such as GHC, which may only report a subset of the errors present in the code or stop at the first error they encounter. Goanna, however, always thoroughly identifies all type errors in the codebase. In the example of Fig. 5 and Fig. 6, Goanna discovered the two errors included in the file. Clicking on the error selector on the bottom left will change the content of the debugging panel and text editor highlights to reflect the cause of a different error (Fig. 7).

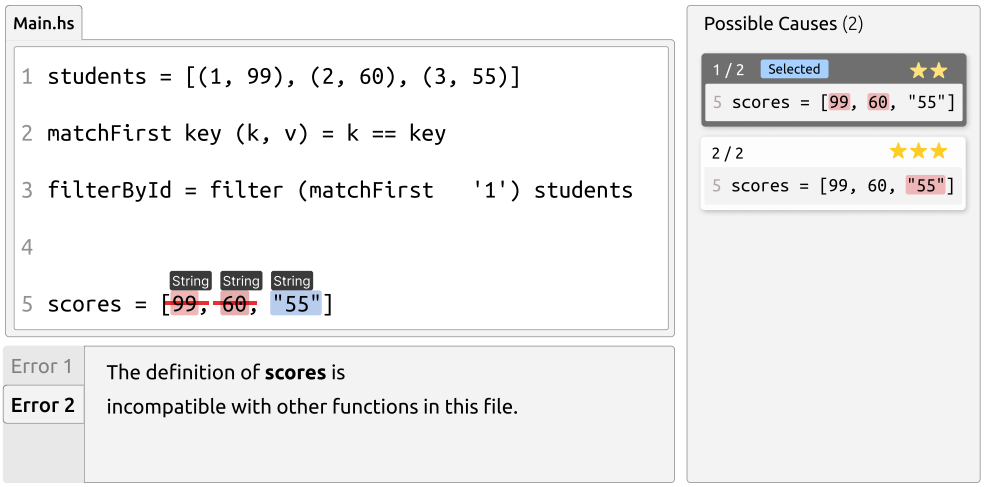

Fig. 7 Selecting a different error in Goanna. Selecting a different error using Goanna's error selector. The debugging panel will show potential cause locations for the selected error. The highlights and type hints in the editor panel will focus on the selected error.

4.3 Type Error Grouping

In addition to reporting multiple errors, Goanna also groups together type errors that might be treated as separate by other tools. Goanna uses a novel approach (section 5.3) to ensure that type errors that are intuitively connected are grouped together. This means that Goanna does not overwhelm the programmer with an excessive number of redundant type errors. Instead, the programmer is presented with a concise list of errors that can all be assessed separately.

For instance, in Fig. 8, the functions `variance` and `mean` expect the final type of `size` to be a fractional value. However, the definition of `size` results in an integral value, which creates a conflict. While GHC shows three separate type errors, Goanna groups these interconnected errors into a single entity, as shown in Fig. 9. These errors can be addressed collectively, thereby improving the efficiency of the programmer.

5 Goanna Implementation

Goanna consists of three phases: constraint generation, MUS enumeration, and post-analysis, each with its distinct role. In the constraint generation phase (Section 5.1), Goanna translates program syntax into constraints. During the MUS enumeration phase (Section 5.2), Goanna identifies all critical constraint subsets, which include Minimal Unsatisfiable Subsets, Minimal Correction Subsets, and Maximal Satisfiable Subsets. Finally, in the post-analysis phase, Goanna applies various optimization techniques and transforms the critical constraint subsets into intuitive type error diagnostics. This phase includes type error grouping (Section 5.3) and case ranking (Section 5.4).

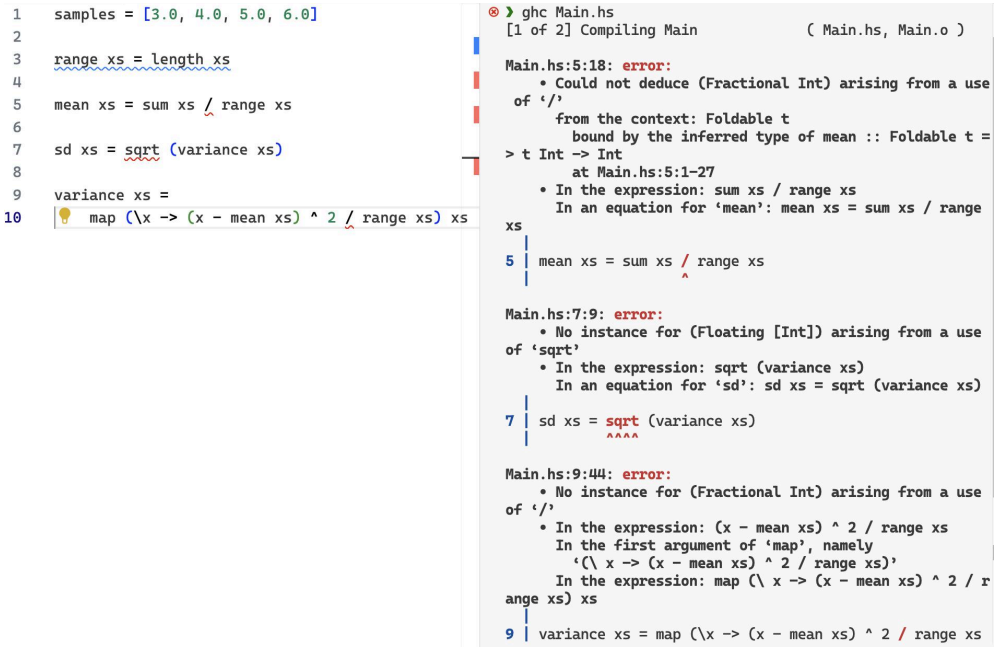

Fig. 8 Inspecting a defective Haskell Program (left) in relation to the error messages output by the standard GHC compiler (right) – 3 separate type errors are reported. The editor (VS Code is used here) underlines the error locations reported in the messages, but all other contextual information must be understood from the error text.

Goanna supports a wide and growing range of the Haskell 2010 language syntax [44]. When writing, fully supported features include module import/export, qualified imports, import hiding, do notation, algebraic data types, new types, type synonyms, type classes, operator sectioning, and range expressions. Only operating at the type level, Goanna automatically supports the language extensions that are not type-related (RecordWildcard) and those that do not extend the rules of the underlying type system (InstanceSigs). Overall, Goanna supports the most widely used type-level extensions in Haskell, and the range of supported features continues to grow. A detailed and updated feature coverage [45] is publicly available.

5.1 Constraint Generation

In this phase, Goanna translates Haskell source code into constraints. Specifically, Goanna uses any ISO-compatible Prolog [46] as its underlying constraint language and solver. Prolog is chosen because it is a widely used logic programming language; this helps generalize Goanna's approach to different programming languages in the future. It also makes the intermediate representation of types readable to a human user with Prolog literacy, a helpful feature to help "visualize" the type inference process and identify errors in the implementation. At the end of the constraint generation phase, a Prolog program is produced. This Prolog program will succeed upon execution if

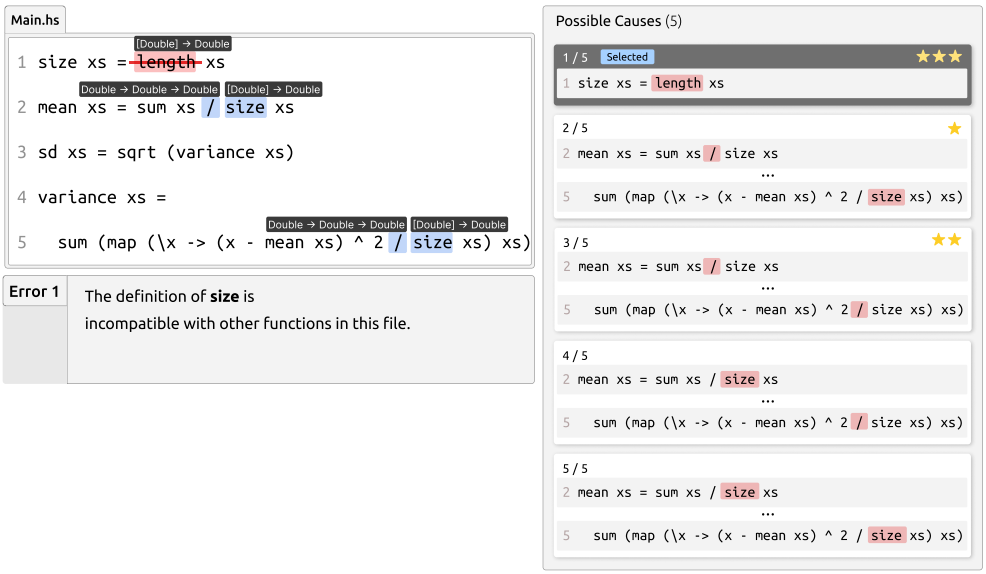

Fig. 9 Goanna's Error Grouping. This error, although its potential offending parts appear in many declarations, is possible to fix in one place, i.e., by changing the definition of the `size` function on line 1. Therefore, Goanna reports it as a single error.

the Haskell code is well-typed and will fail otherwise. Each line of the Prolog program is annotated with a corresponding location in the Haskell source code. During the MUS enumeration phase, these lines can be conditionally activated or deactivated to determine whether a particular location in the Haskell code contributes to a potential issue. This section outlines how we generate this Prolog program.

For a simplified Haskell syntax shown in Fig. 10.A, we generate a list of Prolog predicates in the language shown in Fig. 10.B. We use 4 auxiliary functions during the constraint translation process (Fig. 10.C) to generate Prolog variables for future unification. `fresh` makes a unique unbound Prolog variable. `var` takes a Haskell identifier name and returns a Prolog variable. Naively, this can be done by turning it to uppercase. `atom` takes a Haskell type constant/constructor name and returns a Prolog atom. Naively, this can be achieved by turning it into lowercase. `append_clause` takes a Prolog predicate P and stores it into the set of all generated Prolog predicates $\mathcal{P}$. To clarify, parsed Haskell syntax is in blue, while generated Prolog syntax is in red.

Let Γ be an open-ended list of Prolog atoms containing the names of Haskell local variables. Two sets of generation rules are defined to convert different Haskell syntax nodes to Prolog text. Predicate generation rules (Fig. 10.D) take Γ and Haskell declarations as input and output Prolog predicates. For example, a Haskell function `f = 2`, Goanna may generate a predicate `f(V, _) <- V = int.`

Constraint generation rules (Fig. 10.E) take Γ, a Haskell expression node or type node, and a Prolog variable `V` as input, and output a list of Prolog terms. The evaluation of these terms will attempt to infer a type for the Haskell syntax node, and unify the type with the Prolog variable `V`.

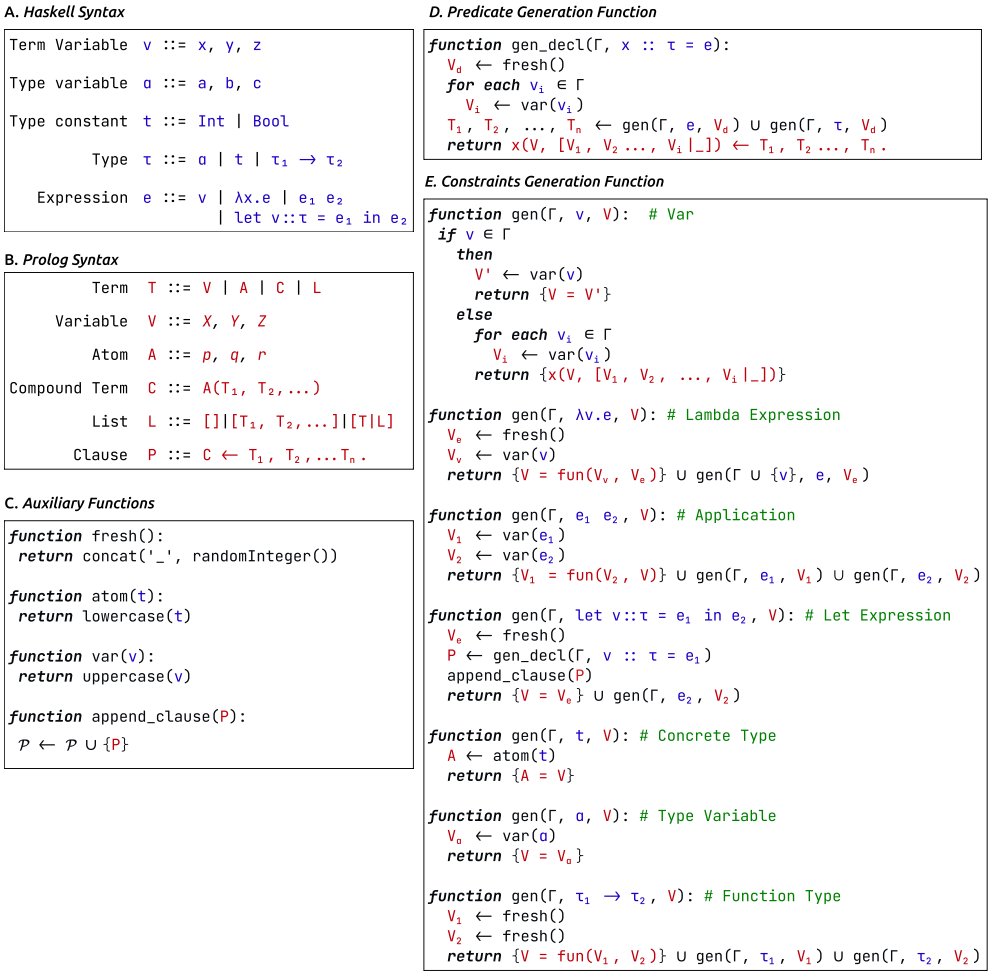

Fig. 10 Goanna's Constraint Translation Rules (Simplified)

An example of such translation can be found in Fig. 11. We use standard Prolog notation `name/arity` (for example *unify/2*) here when referring to a Prolog predicate, as a Prolog predicate is identified by the combination of both attributes. In the Haskell program (Fig. 11.A), 2 functions are declared: `f` and `g`. This will generate two corresponding Prolog predicates `f/2` and `g/2`. In the actual implementation of Goanna, the generated predicates would be `f/6` and `g/6`. The extra arguments are

added to perform various tasks involving syncing state, such as breaking recursive calls and collecting type class constraints. In a predefined predicate `type_check/0`, the subgoals `f(_,_)` and `g(_,_)` are added. Executing the top-level goal `type_check` in a Prolog environment will get a result of `false`.

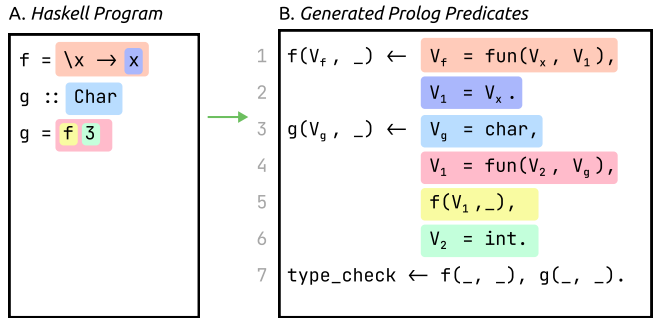

Fig. 11 An example of Goanna constraint generation. For the Haskell functions `f` and `g`, Goanna generates the predicates `f/2` and `g/2`. Each subgoal of `f/2` and `g/2` is generated from a corresponding part of the Haskell program. In a predefined predicate `type_check/0`, the subgoals `f(_,_)` and `g(_,_)` are added. Running the goal `type_check` will return whether the program is well-typed. In this particular example, this will return `false`. We used standard Prolog notation `name/arity` here when referring to Prolog predicates, as a Prolog predicate is identified by the combination of both attributes.

5.2 MUS enumeration

After the constraint generation phase, Goanna uses the MARCO algorithm [33] to efficiently find all MUSes, MCSes, and MSSes. Naively, each MCS corresponds to a potential cause of the type error. The source locations encoded in each constraint of MCS are reported as the offending code that needs to be modified. For the example in Fig. 12, Goanna's MUS enumeration system identifies 2 MUSes, 3 MCSes, and 3 MSSes. Following the 3 MCSes, Goanna reports 3 potential causes of the type error: the type annotation and function definition in `f` (from MCS_1), the type annotation alone in `g` (from MCS_2), and the function definition alone in `g` (from MCS_3).

5.3 Type Error Grouping

Sometimes, it is insightful to inform programmers that multiple type errors are related to each other (caused by the same violation of typing rules) or isolated instances. To achieve this, Goanna introduced a novel feature: type error grouping. When knowing all the MUSes, Goanna can analyze how many isolated type errors there are and what locations are involved in each type error. The algorithm for performing type error isolation is as such:

Let U denote the set of all Minimal Unsatisfiable Subsets (MUSes) and C the set of all Minimal Correction Sets (MCSes). We define an undirected graph G, where each vertex in G corresponds to a minimal unsatisfiable subset $u_i \in U$, and the edges of G

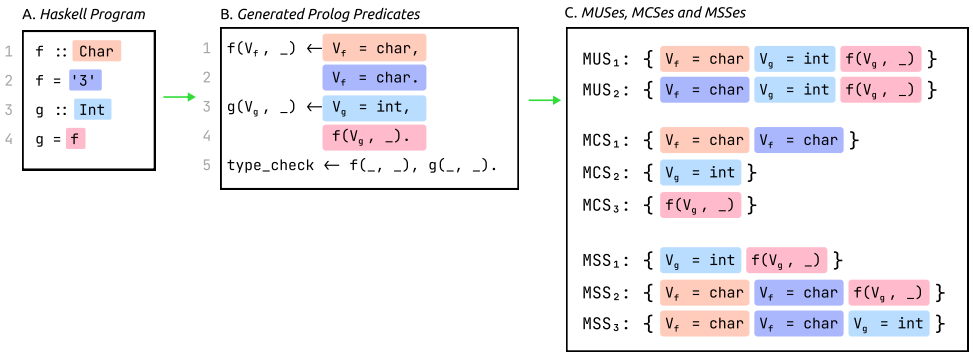

Fig. 12 An example of Goanna MCS Enumeration. From the set of constraints (B) generated from the Haskell program(A), Goanna obtained 2 MUSes, 3 MCSes, and 3 MSSes.

connect pairs of MUSes u_i and $u_j \in U$ if their intersection is non-empty. The set of all connected components D in G represents the set of all type errors. For each $d_i \in D$, let $l_i = \bigcup v_i$, where v_i is the set of vertices in d_i. l_i is the set of all constraints local to this type error. Define $C_i = \{x \mid \forall c \in C, x = c \cap l_i\}$ as the set of all MCSes that are local to this type error.

This can be intuitively thought of as follows: two type errors can be grouped together if they cannot be fixed independently through modifying a minimal set of locations for each. For instance, Fig. 13.A shows one connected type error, where there are two fixes available: change `0` on line 1 to a Boolean type, change the type annotation on line 3 to a `Num` instance, or change the assignment of `y` to a different expression. Choosing either one will result in both `x` and `y` being inferred to have a valid type.

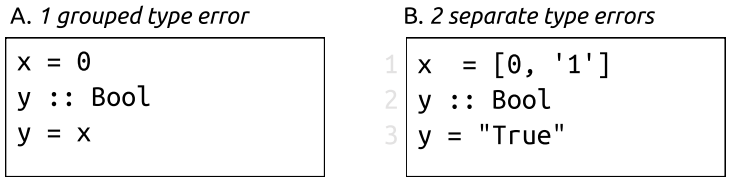

Fig. 13 Goanna's type error grouping. The ill-typed program on the left contains a single type error, because it can be fixed by a minimal set of syntax changes. For example, fix it by changing the literal `0` on line 1 to `True` or `False`. This edit contains a single location, so there exists no smaller edit that can fix `x` or `y` alone. The program on the right contains two type errors because `x` or `y` can be fixed separately. For example change `0` to `'0'` on line 1 fix `x` alone.

However, in Fig. 13.B, although the ill-typed fragment is in a single function, we can fix the argument, either `0` to `'0'` or `'1'` to `1` to eliminate part of the type error. The same goes for the function's result type `x`. In this case, there are two separate type errors that should not be grouped.

In practice, type error grouping provides a sense of the "effective area" of a type error. Programmers are commonly bewildered by the fact that changing one place in the program causes an error in a seemingly unrelated area. When refactoring a known correct program, a programmer can change the definition of one variable, and Goanna will show all the locations that require further changes. This works because all the further changes belong to the same type error, because a single syntax change – reverting the initial changes – will result in the program being well-typed once more.

More specifically, to Goanna, type error grouping provides an effective means to reduce the number of causes. In an ill-typed program with m errors, each having n potential causes, it will result in nm total causes. Dividing these causes into separate errors that align correctly with intuition is the most important technique to enhance Goanna's error reporting.

5.4 Cause Ranking

In Goanna-IDE, when a list of potential causes is on display, Goanna-IDE also shows the top "likely" causes according to the ranking heuristics. This is very helpful because programmers will have a starting point for the investigation. We provide a set of efficient heuristics for ranking suggestions.

Number of Error Locations

Causes comprising fewer locations are prioritized and presented earlier in the list. For instance:

```
1 x a
2   | a < 10 = 'A'
3   | a >= 10 && a < 100 = 'B'
4   | otherwise = "C"
```

Fig. 14 Goanna prefers causes with fewer error locations. In this ill-typed program, Goanna chooses to give the cause `"C"` on line 4 a higher likelihood because it contains a single location. The other cause contains 2.

In the example in Fig. 14, there exist two possible fixes: 1) Changing `'A'` and `'B'` to the string type, and 2) changing `"C"` to the `Char` type. As the latter fix affects only 1 location (as opposed to 2 in the former), it is assigned a higher ranking and appears earlier in the list.

Change Specificity

Another useful heuristic is encouraging the cause whose fix will result in every surrounding term being as concrete as possible.

In the example in Fig. 15, Goanna can suggest two potential causes and fixes. First, change the integer literal `3` to a `Bool` type. Second, change the function `not` to a different function that accepts an integer as input. The second fix results in variable

```
1 not True = False
2 not False = True
3 x = not 3
```

Fig. 15 Goanna prioritize causes whose resolutions lead to more concrete type assignments. In this example, change `3` will result in `x` to have type `Bool`. Alternatively, `x`'s type will be unknown after changing `not` on line 3. Goanna prefers the former.

`x` having a less concrete type. Indeed, `x` can have any type if we don't limit what function to replace `not` with. Goanna prioritizes the first cause over the second.

6 Evaluation

In evaluating Goanna as a type error debugging environment, we aim to address the following key research questions:

1. **RQ1** Can Goanna identify type errors more accurately compared to traditional tools?
2. **RQ2** How fast can Goanna provide these type error diagnoses?

To answer **RQ1**, we conducted a corpus study on defective Haskell programs compiled from online channels and research papers. For **RQ2**, we performed benchmarks using generated Haskell programs with controlled properties.

6.1 The Datasets

We compiled two datasets of defective Haskell programs (N=153) for running the evaluation. The first dataset (N=86), named `Haskell Community`, was sourced from Haskell online discussion forums, each containing one or more type errors. These include StackOverflow, Reddit, and Discord, which are the top communications channels for Haskell users based on a research survey [47]. We focused on discussions where authors encountered type errors and sought help, including only those with solutions accepted by the original authors. The second dataset (N=67), called *Haskell Academic*, was compiled from 23 peer-reviewed research papers that are related to Haskell type errors. This include type system features [26, 48–50], systems to improve type error [15, 17–19, 27, 36, 37, 39, 51–56], corpus studies [14, 57, 58], tool demonstration [16, 59].

Program lengths ranged from 1 to 64 lines of code (mean=20, median=20). These programs covered a variety of themes, including basic syntax (14 files), lists (28 files), tuples (5 files), algebraic data types (22 files), higher-order functions (17 files), monadic operations and do notation (9 files), type classes (6 files), and built-in/library functions (24 files). This thematic distribution aligns with the different causes of type errors from Tirronen's study [14].

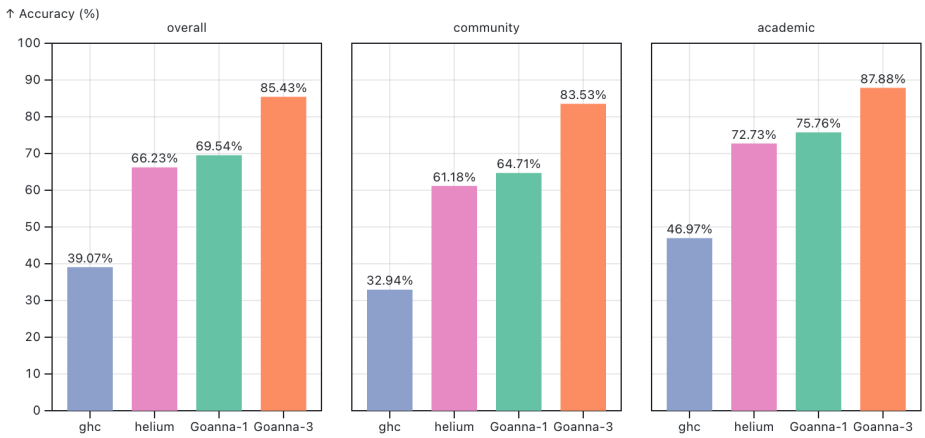

Fig. 16 Accuracy of GHC, Helium, Goanna-1, and Goanna-3

6.2 Accuracy and Reliability

We define accuracy in type error diagnosis as whether the reported error aligns with the user's real intention. On the other hand, reliability is indicated by whether the type error can be resolved by making suitable changes to the reported location. An error diagnosis can be reliable but inaccurate. For each defective program in our datasets, we compare Goanna's error diagnosis against the Oracle answer to measure the overall accuracy. A diagnosis is deemed accurate if its suggested fix matches the accepted answer in all reported locations. It is considered reliable if the type error can be resolved when the reported locations are changed to values of the bottom type (*undefined* in Haskell).

We compared Goanna with the Glasgow Haskell Compiler (GHC) version 9.4 [60] and Helium [61] version 1.8. GHC is the most widely used Haskell compiler, and it is known for its comprehensive capabilities and efficiency. Helium is another well-regarded Haskell compiler noted for producing high-quality error messages [16]. Other candidates include Counter-factual Typing (CFT) [15, 36, 37] and SHErrLoc. However, these tools are not being included in comparison because their executables are not easily reproduced. Because Goanna suggests multiple potential causes, it technically always includes the accurate cause in its diagnostics. Therefore, we only included two variations in this comparison: Goanna-1 (including only the top suggestion) and Goanna-3 (including the top three suggestions).

Accuracy: As seen in Fig. 16, GHC, with an accuracy of 39.07%, is the lowest among the tools. Helium demonstrates meaningful improvement over GHC, with an accuracy of 66.23%. Goanna-1 outperforms both GHC and Helium with an accuracy of 69.54%. **Goanna-3 achieves the highest accuracy at 85.43%**, maintaining a consistent level across both the *Haskell Academic* and *Haskell Community* datasets, while other tools see drops in accuracy in *Haskell Community*. For all errors in the

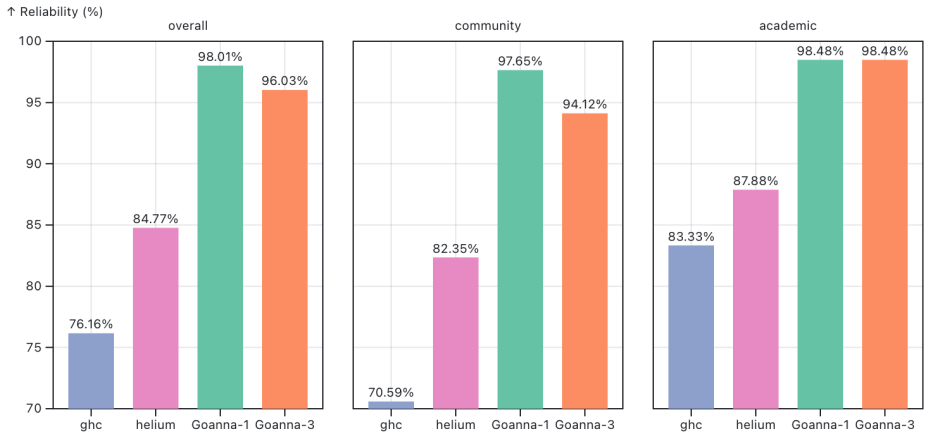

Fig. 17 Reliability of GHC, Helium, Goanna-1, and Goanna-3. The figure is cropped from 70% to 100% for readability.

datasets, there were no disagreements among the tools regarding the presence of errors; thus, Goanna did not miss any errors identified by other tools nor falsely identify a well-typed program as defective.

Reliability: As shown in Fig. 17, GHC is the least reliable (76.16%), while Helium has modestly better reliability (84.77%). **Goanna-1 and Goanna-3 both achieve the higher reliability of 98.01% and 95.3% respectively**. In this, GHC and Helium are both unsuccessful when multi-witness type errors and multi-party type errors occur. Both of these tools report only a single location. Helium performs better than GHC because Helium sometimes reports errors on a larger syntax block, such as a list expression or a function application. This larger syntax block is often general enough to encompass the entire multiple error locations in the leaf nodes. GHC tends to report on leaf nodes and often results in less reliable error diagnostics. Note that Goanna-3's reliability can be worse than Goanna-1. This happens because of the cases when Goanna-3 includes a cause that cannot lead to a successful resolution in its second or third ranking cause.

Example: Figure 18 illustrates the differences among GHC, Helium, and Goanna when analyzing the erroneous function `fac`. In this instance, GHC generated an error message that identified an incorrect location and was misleading. GHC suggested that implementing an instance of `Num Bool` would resolve the issue, which is not the case, displaying a lack of both accuracy and reliability. On the other hand, Helium's error report highlighted the general area of the root cause but failed to localize the issue, which was a misspelled operator. However, following Helium's suggestion to replace the expression `n == 1` with an expression of type `Int` indeed resolved the error, demonstrating reliability in this instance. Goanna's top recommendation, Goanna-1, accurately identified the location of the issue and provided a correct typing suggestion,

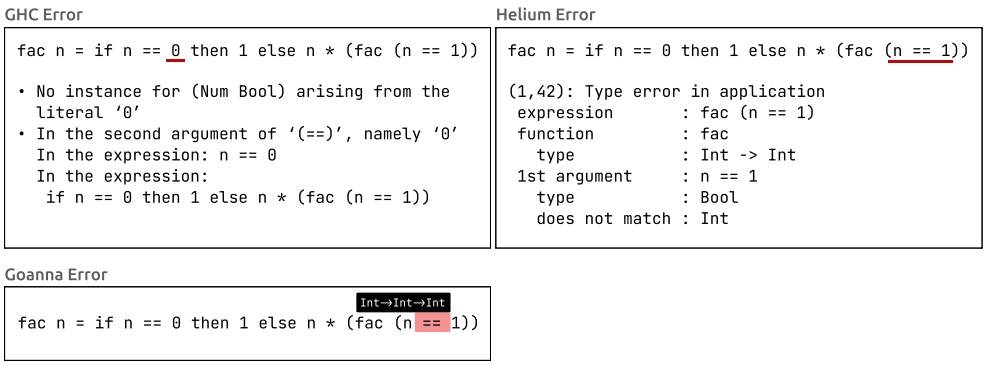

Fig. 18 Comparison of Type Error Reports from GHC, Helium, and Goanna-1. This example involves an incorrectly defined function `fac`, where the programmer erroneously wrote `n == 1` instead of `n - 1` in the function's definition. Goanna accurately identified the precise location of the error. In contrast, Helium located a broader area, identifying the parent node. GHC, however, indicated a location unrelated to the root cause and provided a misleading error explanation.

recommending the replacement of == with a function of type `Int -> Int -> Int`. Goanna-1 was both accurate and reliable in this example.

6.3 Performance

Goanna's high accuracy and reliability come at the tradeoff of its computational complexity. To understand how Goanna performs compared to other tools, we recorded the time required for all 3 tools to generate type error diagnostics for each program in our datasets. Since Goanna continues until all causes are detected and ranked, there is no distinction between Goanna-1 and Goanna-3 with respect to performance. As we can see in Fig. 19 when measuring the time required to generate full type error diagnoses on our datasets, Goanna is slower than other tools: GHC (mean=0.09s, median=0.09s), Helium (mean=0.31s, median=0.31s), and Goanna (mean=0.50s, median=0.42s). Additionally, Goanna exhibits a larger variation in performance.

The computational expense of Goanna is primarily attributed to Minimal Unsatisfiable Subset (MUS) enumeration, which is constrained by two key factors. The first factor is the cost of identifying a single MUS, determined by the size of the program. The algorithm for finding an MUS involves the repeated removal of a single random constraint followed by satisfiability testing, a process known as "*shrink*". Since the number of constraints corresponds to the number of syntax nodes in the program, longer programs require more satisfiability checks, resulting in a longer "*shrink*" process. The second factor is the total number of MUSes, which depends on the category (Section 3) of the type error in question. Consequently, we analyzed performance based on program size and type error category.

Program Size:

To measure the impact of program size on Goanna's performance, we constructed four sets of programs: Set 1 contains a type error involving 2 locations, Set 2 contains a type

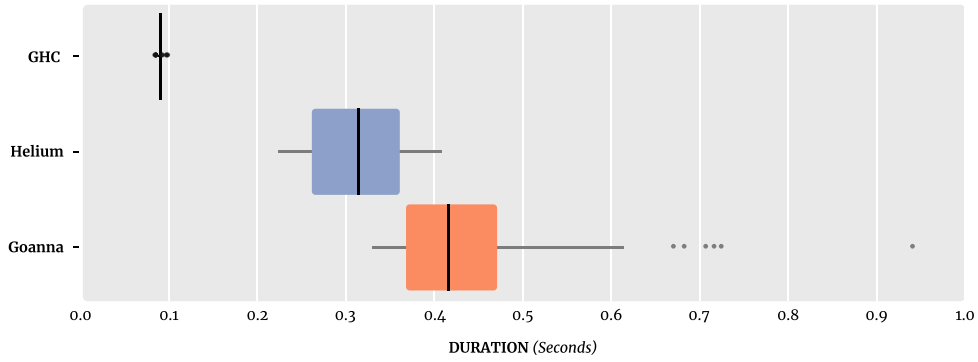

Fig. 19 Goanna's performance compared to GHC and Helium. GHC is the fastest among all three (mean=0.09s, median=0.09s), followed by Helium (mean=0.31s, median=0.31s). Goanna performed slower than other tools (mean=0.50s, median=0.42s).

error involving 10 locations, Set 3 contains a type error involving 20 locations, and Set 4 contains a type error involving 30 locations. For each set, we initiate testing with the minimal number of syntax nodes and progressively increase the program length to contain up to 1200 syntax nodes, while the attributes of the type error remain unchanged. Fig. 20 demonstrates the impact of program size on the time taken to generate type error diagnostics. It shows that Goanna's performance is slowed down with the increase in the size of the program; this holds even when the type error remains unchanged. The diagnosis takes 0.25 seconds with 200 syntax nodes for a type error involving two potential fix locations. Still, it increases to 7 seconds with 1200 syntax nodes. Furthermore, it shows that more complex type errors require additional time for diagnosis, independent of the effect of program size.

It is worth clarifying that performance bottlenecks are primarily due to MUS enumeration rather than parsing or static analysis, as these processes are negligible for reasonably sized programs. Also, having more files in the project and using third-party libraries **do not contribute to the computation overhead**. Although those files are translated into constraints, they are considered as basic assumptions and do not participate in the *shrink* process.

Multi-step Type Errors

To better understand the effect of multi-step type errors on performance, We generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-step type error comprising 3 steps and gradually increase up to 30 steps, maintaining constant program size. Fig. 21 details the methods of generating multi-step type errors with various numbers of steps while maintaining the constant program size.

Fig. 22 plots the time required to generate type error diagnostics as a function of the number of steps involved in a type error. It can be seen that the impact of program

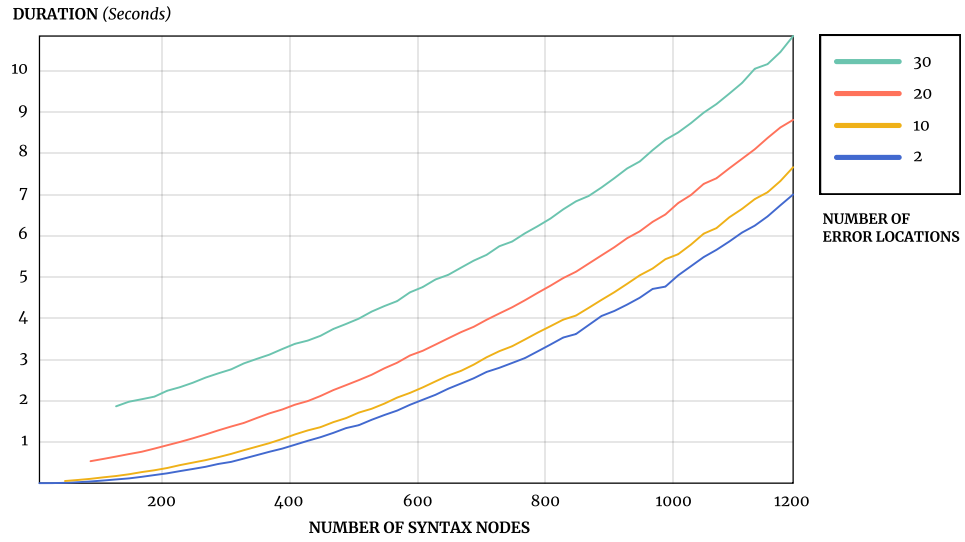

Fig. 20 Goanna's Performance with Respect to Program Size. To evaluate the impact of program size on Goanna's performance, we generated four sets of programs: Set 1 contains a type error involving 2 locations, Set 2 contains a type error involving 10 locations, Set 3 contains a type error involving 20 locations, and Set 4 contains a type error involving 30 locations. For each set, we initiate testing with the minimal number of syntax nodes and progressively increase the program length to contain up to 1200 syntax nodes. This figure illustrates the time taken to generate type error diagnostics as a function of the number of syntax nodes for each set. The results indicate that the time required escalates with the size of the program, even when the type error remains consistent. Furthermore, it shows that more complex type errors require additional time for diagnosis, independent of the effect of program size.

size is independent of the effects of the number of steps in the errors. For a program with 200 syntax nodes, it takes 0.275 seconds to diagnose; with 30 steps, it requires 2.8 seconds. For the program with 1000 syntax nodes, 2-step errors take 5 seconds, and 30-step errors take 9 seconds.

Multi-witness Type Errors

Similar to evaluating multi-step type errors, we generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-witness type error with 1 witness on one side of the type error and 3 on the other, denoted as $(1, 2)$. We gradually increase the number of witnesses up to $(1, 30)$. Fig. 23 details the methods of generating multi-step type errors with various numbers of steps while maintaining the constant program size.

Fig. 24 illustrates the time required to generate type error diagnostics with respect to the number of witnesses involved in a type error. With 30 error locations and 200 syntax nodes, diagnosis takes a similar time as multi-step errors. With 1000 syntax nodes, it takes over 43.68 seconds. In addition, we can visually identify the linear

Steps = 3, *Nodes* = 24

```
x :: (Int, Char)
x = (a, b)
  where a = '3'
        b = c
        c = '3'
```

Steps = 4, *Nodes* = 24

```
x :: (Int, Char)
x = (a, c)
  where a = b
        b = '3'
        c = '3'
```

Steps = 5, *Nodes* = 24

```
x :: (Int, Char)
x = (a, '3')
  where a = b
        b = c
        c = '3'
```

Error Locations

Fig. 21 Example of generating multi-step type errors with various numbers of steps. In this example, we generated 3 Haskell programs, each containing a multi-step type error. The type error in program 1 (left) contains 3 steps; the one in program 2 (middle) contains 4 steps; the one in program 3 (right) contains 5 steps. All 3 programs contain 24 syntax nodes. The second element of the 2-tuple is used to balance the length of the program.

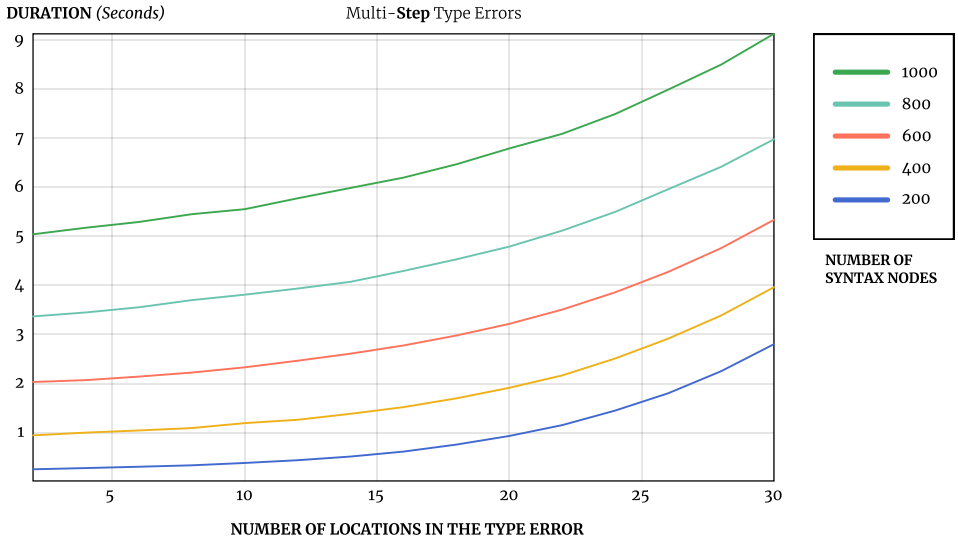

Fig. 22 Impact of Multi-step Type Errors on Performance. We generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-step type error comprising 3 steps and gradually increase up to 30 steps, maintaining constant program size. This figure illustrates the time required to generate type error diagnostics with respect to the number of steps involved in a type error.

growth in computation time because we only increase the number of witnesses on one side. Recall that for a type error that involves m witnesses on one side and n on the other, there will be a total of mn MUSes.

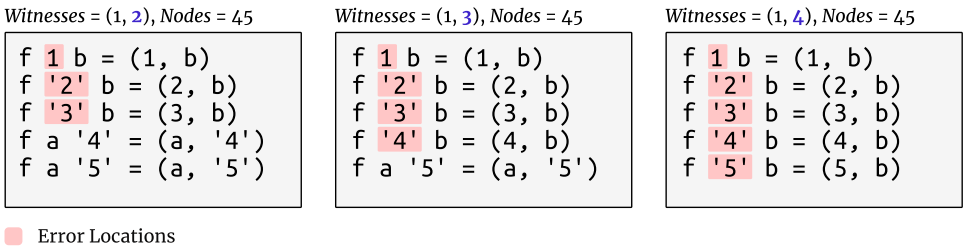

Fig. 23 Example of generating multi-witness type errors with various numbers of witnesses. In this example, we generated 3 Haskell programs, each containing a multi-witness type error. The type error in program 1 (left) contains 1 and 2 witnesses on each side, denoted as $(1, 2)$; the one in program 2 (middle) contains $(1, 3)$ witnesses; the one in program 3 (right) contains $(1, 4)$ witnesses. All 3 programs contain 45 syntax nodes. The second argument of function f and the second element of the 2-tuple are used to balance the length of the program.

Multi-party Type Errors

Multi-party type errors are often multiple simple errors fused into one and are the most computationally intensive. Again, we generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-party type error comprising 3 parties and gradually increase up to 30 parties, maintaining a constant program size. Fig. 25 details the methods of generating multi-step type errors with various numbers of steps while maintaining the constant program size.

Our results (Fig. 26) show that, for 30 error locations and 200 syntax nodes, diagnosis requires 3.15 seconds; with 1000 nodes, it takes over two minutes. Recall that for a multi-party type error with n parties, there are $n(n-1)/2$ unique MUSes. The growth pattern in computation time roughly reflects this correlation.

Note that this experiment does not mean to be representative of realistic type errors. It is often improbable to have more than 3 parties in real-world programming; for more than that, programmers have to ignore an existing type error and continue adding more ill-typed code fragments.

Performance Summary

The size of the program influences Goanna's performance, as do the category and complexity of errors. Specifically, multi-party errors pose the greatest computational challenge among the various categories, followed by multi-witness and multi-step errors (see Fig. 27). We showed that this performance constraint is approximately correlated with the number of Minimal Unsatisfiable Subsets (MUSes) in the type error. Following this insight, we discuss several avenues to improve the performance of Goanna and systems using similar techniques in Section 7.2.

It is important to note that our benchmarking aims to evaluate the performance characteristics of Goanna under stress conditions. Some of the samples we generated are unlikely to appear in typical programming scenarios. Generally, common type errors in smaller programs (fewer than 100 lines of code) can be resolved by Goanna

Fig. 24 Impact of Multi-witness Type Errors on Performance. Similar to evaluating multi-step type errors, we generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-witness type error 1/3 witness, that is, 1 witness on one side of the type error and 3 on the other, and gradually increase up to 1/30 witnesses. This figure illustrates the time required to generate type error diagnostics with respect to the number of witnesses involved in a type error.

within approximately one second, within the waiting time tolerance for real-time feedback to programmers [62]. All defective programs in our datasets are diagnosed in under 1 second. However, Goanna demonstrates moderate performance when diagnosing larger programs (exceeding 100 lines) or handling extreme type errors, such as multi-party errors involving numerous parties. In such cases, it is more practical for Goanna to offer asynchronous, on-demand diagnosis rather than real-time feedback.

7 Discussion

We discuss Goanna's accuracy and reliability, along with its practical advantages over traditional tools. We then analyze the trade-offs made in Goanna's performance and explore avenues to improve on this limitation. Finally, we compare the strengths and weaknesses of Goanna with Large Language Model (LLM)-based programming assistance and explore how these two approaches can be integrated for optimal benefits.

7.1 Strengths of Goanna and MCS-based Error Diagnostics

Goanna is the first debugging tool to use Minimal Correction Subsets (MCS). Compared to traditional methods like Hindley-Milner and constraint-based type inference using Minimal Unsatisfiable Subsets (MUS), Goanna provides a more concise cause

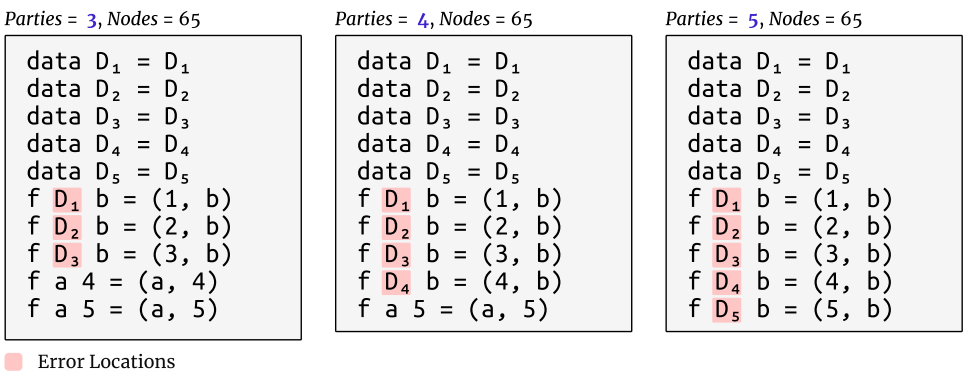

Fig. 25 Example of generating multi-party type errors with various numbers of witnesses. In this example, we generated 3 Haskell programs, each containing a multi-party type error. The type error in program 1 (left) contains 3 parties; program 2 (middle) contains 4 parties; program 3 (right) contains 4 parties. All 3 programs contain 65 syntax nodes. The second argument of function f and the second element of the 2-tuple are used to balance the length of the program.

analysis and better guesses of root causes. It identifies all type errors in the source file rather than just one and infers the most concrete types for the entire source program.

Remove Biases in Type Error Reporting

One factor that makes Goanna surpass traditional tools in accuracy is its unbiased approach toward all possible causes. Conventional compilers and type checkers, such as GHC and Helium, terminate upon finding the first type error, resulting in only the usage of a function being reported in the error message. At the same time, its definition is never blamed, an issue known as left-to-right bias in traditional type-checking algorithms, a notable shortcoming that limits the usability of error messages [50, 63, 64].

Unlike conventional compilers, which add constraints one at a time, Goanna applies a holistic approach by collecting all constraints from the source program and delaying the solving until the entire program has been translated into a constraint-level intermediate representation. This approach allows Goanna to assert multiple assumptions regarding the type of error using constraint programming techniques. In practice, reporting multiple possibilities for an error is highly useful. Programmers are equally prone to making mistakes in both calling and defining functions. It has been noted that some common causes of type errors among Haskell programmers occur at the definition site, such as outdated type annotations.

Meaningful and Familiar Idioms to Better Understand Type Errors

Goanna uses MCS as the building block of a type error diagnosis. This choice allows type errors to be presented as a list of possible causes; each cause shows the smallest set of changes required to fully resolve the error. Goanna is not unique in employing this paradigm of debugging errors. This type of error diagnosis has been utilized in various user interface designs, such as spell-check tools. Additionally, Goanna allows

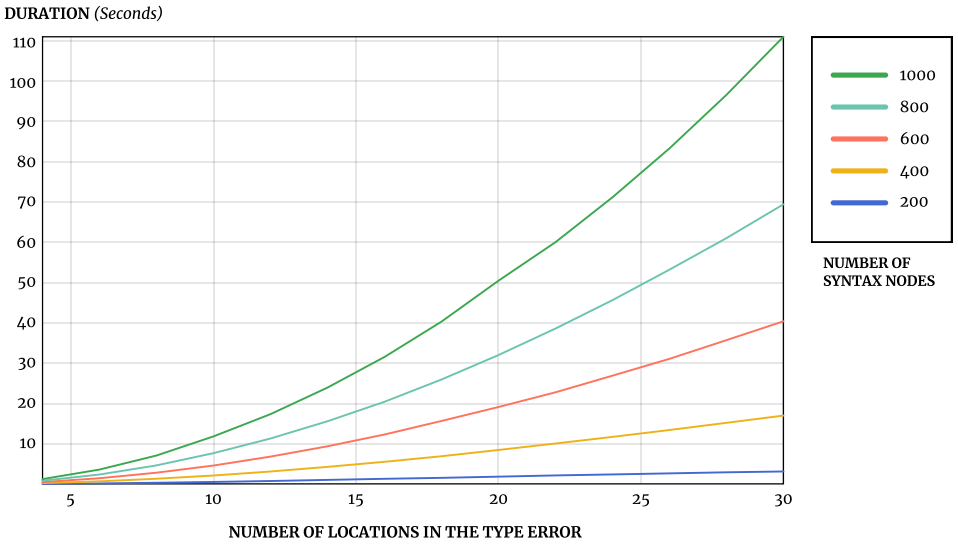

Fig. 26 Impact of Multi-party Type Errors on Performance. We generated five sets of programs differentiated by program length: Set 1 consists of 200 nodes, Set 2 consists of 400 nodes, Set 3 consists of 600 nodes, Set 4 consists of 800 nodes, and Set 5 consists of 1000 nodes. For each set, we initiate with a multi-party type error comprising 3 parties and gradually increase up to 30 parties, maintaining a constant program size. This figure illustrates the time required to generate type error diagnostics with respect to the number of parties involved in a type error.

programmers to preview each possible cause: when navigating to a cause, locations that need to be changed are highlighted, and the effect of these changes on the program semantics and types is reflected through dynamically updated type hints. This interface idiom is analogous to merge conflict resolution tools in modern version control systems.

Full Type Hints on Ill-typed Program

In statically typed languages, particularly those like Haskell, where expressions are implicitly typed (type annotations are optional), understanding the types of various expressions is an integral part of making sense of the program. Traditional approaches run in interactive shells or command lines, such as `:t expression` command the interactive shell for Haskell (Fig. 28 Left). Modern editors enhance this functionality by providing text overlays for instant type overview (Fig. 28 Right). However, these type hints are available only when the program is well-typed; they either become unavailable or outdated when type errors occur. This limits their usefulness, as they are much more needed when encountering type errors.

Goanna addresses this issue by inferring types for ill-typed programs to the greatest possible extent using the Maximal Satisfiable Subset (MSS). MSSes are the complement set to MCS. Therefore, Goanna has access to an MSS for each MCS. The intuition is that MSS represents the largest portion of the source code that is well-typed, allowing a set of type assignments for each possible cause. That is, Goanna's type hints

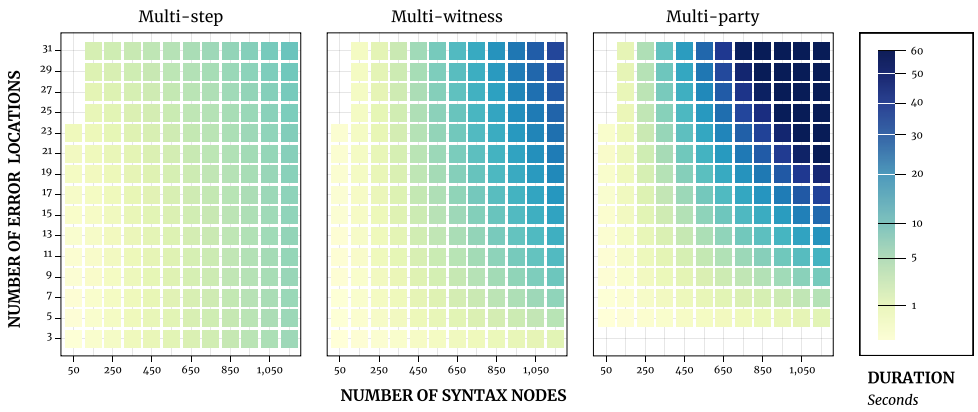

Fig. 27 Performance Comparison Across Three Categories of Type Errors. The x-axis indicates the number of syntax nodes in the program, whereas the y-axis signifies the number of locations involved in the type error. In the case of multi-step type errors, the number of locations corresponds to the number of steps; for multi-witness type errors, it corresponds to the number of witnesses; and for multi-party type errors, it represents the number of parties. The color scale indicates the time required to generate type error diagnostics, which is represented on a square-root scale to more accurately reflect the computational time increase.

```
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :t fmap
fmap :: Functor f ⇒ (a → b) → f a → f b
ghci> :t (reverse [1..10])
(reverse [1..10]) :: (Num a, Enum a) ⇒ [a]
ghci>
```

```
1  data BinTree = Leaf | Label Int BinTree BinTree
2
   depth :: (Num a, Ord a) => BinTree -> a
3  depth Leaf = 0
4  depth (Label a l r) = 1 + max (depth l) (depth r)
5
   isEven :: (Eq a, Num a) => a -> Bool
6  isEven 0 = True
7  isEven n = isOdd (n - 1)
```

Fig. 28 Different Approaches to Querying Types in Haskell. In the Glasgow Haskell Compiler Interactive (GHCI), users employ the command `:t expression` to query type information. Modern integrated development environments (IDEs), such as Visual Studio Code, intrinsically display this type-level information inline within the code, thereby eliminating the need for programmers to make manual type queries.

are two-dimensional: they encompass every expression in the program and also reflect how types would change if a different cause for a type error is selected. This additional information helps verify whether a Goanna's diagnosis aligns with the programmer's true intentions, providing more confidence in identifying the root cause.

7.2 Optimizing Performance

As we preluded in Section 6, Goanna's performance is constrained by the computational cost of the MUS enumeration. More specifically, the performance is limited by the efficiency in discovering a single MUS through the shrink algorithm and the total number of MUSes present in the program.

One potential solution to eliminate the first limitation is to reduce the scope constraint solving from each file into smaller units, such as top-level function definitions, and process these sequentially in topological order. Goanna chose to use a single file as the smallest type-checking unit to provide a concise and minimal formalization of the constraint-based type system. Smaller source code units result in a smaller constraint system, which profoundly mediates the pressure of MUS enumeration. This method is widely used in mainstream compiler tools to manage type-checking in real-world programs.

To address the second limitation, a feasible approach is to explore different MUS enumeration algorithms. The parallel capabilities of advanced MUS enumeration methods [34, 35] have not been fully explored in this study. In addition, partial MUS enumeration [33, 65] can be employed to provide early results to the programmer instead of waiting for the complete list of possible causes.

7.3 Comparison with Large Language Models

LLMs were actively experimented with performing programming-related tasks such as writing code, finding logical errors, and generating documentation. This development is tangential to Goanna's objectives of explaining type errors and supporting easier resolution of type errors.

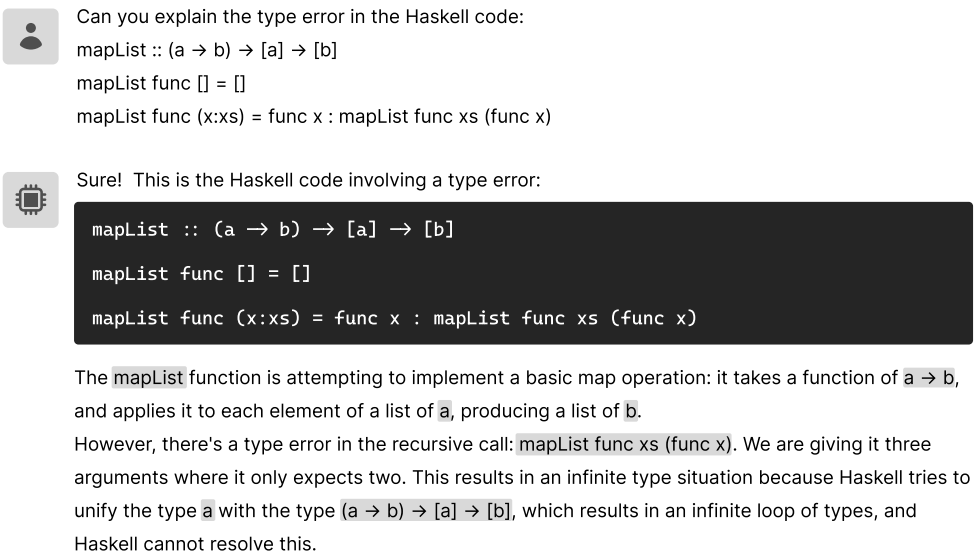

Fig. 29 The figure shows the user prompting LLM to explain the type error in a Haskell function definition `mapList`. LLM, at first, showed a profound understanding of the subject matter, clearly explained the intention of the function, and accurately identified the error location. In the last sentence, LLM claimed the definition would result in an infinite type because of trying to unify `a` with `(a -> b) -> [a] -> [b]`. In reality, unifying two functions with different numbers of arguments will not cause infinite types.

While LLMs have shown capability in recognizing basic errors, they often display a limited understanding of type theories and reasoning ability. Fig. 29 depicts a type error caused by mismatching numbers of arguments in the function `mapList`; although provided a correct error identification at the start, GPT 4 proceeded to claim, "This results in an infinite type situation because Haskell tries to unify the type `a` with the type `(a -> b) -> [a] -> [b]`, which results in an infinite loop of types, and Haskell cannot resolve this," illustrating a fundamental lack of understanding of type theory.

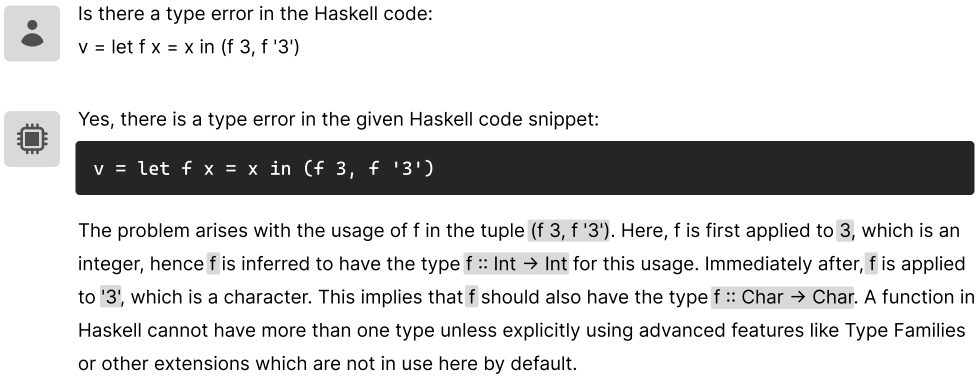

Fig. 30 The figure shows a user asking an LLM whether the provided Haskell source code has a type error. While the provided code is, in fact, well-typed, LLM hallucinated a type error and ignored how polymorphic functions work in Haskell. In the example, the function `f`, defined the same way as the `id` function, can be applied with a value of any type.

When tested with the prompt "Is there a type error in the Haskell code: `v = let f x = x in (f 3, f '3')`", most LLMs incorrectly reported yes (Fig. 30). However, this Haskell code is indeed well-typed. LLMs can give wrong explanations and find type errors that do not exist, unlike tools like GHC, Helium, and Goanna. While there is clearly a role for their usage in programming assistance [66], they do not "reason" about types and, hence, are not trustworthy.

While LLMs are getting more and more accurate with each iteration, some doubt whether they will ever become as reliable as theory-based tools [67]. We believe in the great potential of integrating LLMs with existing theory-based tools, such as Goanna. This integration could enhance the performance of LLMs by aligning their operations with accurate theoretical guides or utilizing their strengths to complement traditional tools' shortcomings, such as suggesting syntactical improvements and coming up with personalized error messages suited for the programmer's skill level. This synergy could lead to more robust programming assistance using both types of tools. We propose two candidate workflows:

- **Code Generation with Validation**: LLMs could generate initial source code based on user-provided prompts. Theory-based tools, such as Goanna, then check

for errors. Detected errors could be fed back to the LLMs for corrections, enhancing the efficiency and accuracy of code development.
- **Error Tutoring and Resolution**: Theory-based tools could identify errors in manually written code, with LLMs suggesting syntax corrections. This combines the precise error detection from traditional theory-based tools with the ability of LLMs to generate effective syntax changes, surpassing traditional tools in this domain.

8 Future Work

Planned User Studies and Research Questions

Our evaluation of Goanna so far relies on existing datasets and program generation. Evaluating Goanna with human participants is crucial for obtaining qualitative insights into its usefulness in explaining and supporting sense-making in complex type errors. We conducted a few preliminary workshops where university students were shown and invited to use Goanna, and most of them reacted positively to using Goanna's debugging experience. However, a rigorous human-centered study based on realistic debugging use cases is planned for the near future. We aim to investigate the following questions:

- Can Goanna's accuracy in type error identification contribute to improved understanding and performance in real-world programming tasks
- Which user interface paradigms effectively leverage Goanna's strengths (such as accessing all possible error causes) and mitigate its limitations (such as slow response times with large programs)?

Suggesting Syntax Approaches

Because Goanna explores the mutation space of types, its suggestions can only provide type-level changes. For example, it may suggest *change this string literal "3" to an expression of type Int*, but not *change this string literal "3" to 3*. An exciting avenue for enhancement is to suggest syntax-level changes that potentially solve the type error. As noted in [15], syntax changes pose significant challenges. However, with recent advancements in generative models and research progress in ML-based type error resolution [51], the development of accurate syntax change suggestions is becoming increasingly feasible, as we discussed in Section 7.3.

Supporting Additional Languages

Expanding Goanna's capabilities to encompass other functional programming languages beyond Haskell is also a consideration. Languages such as Scala and OCaml, which practice similar type systems to Haskell, can directly benefit from Goanna's functionalities. We also intend to extend these techniques to popular multi-paradigm languages like TypeScript and Rust.

9 Summary

In this paper, we introduced Goanna, a tool for identifying and resolving type errors in Haskell, and Goanna-IDE, an interactive Haskell programming environment. We demonstrated the features of Goanna through examples, including finding all possible causes, type error grouping, and identifying multiple type errors. We also discussed our approaches to reduce and reprioritize potential causes.

We evaluated the effectiveness of Goanna on a set of 153 Haskell programs from online sources and research literature. We demonstrated its ability to identify the root causes of type errors accurately and reliably compared to traditional tools. We also evaluated Goanna's performance using controlled benchmarks. We show that Goanna can complete diagnostics in 1 second against small to medium-sized programs. However, Goanna shows modest performance against large programs. Goanna currently works with Haskell, but we plan to extend its MCS-based error diagnostics to work with other strongly typed languages.

References

[1] Ray, B., Posnett, D., Devanbu, P. & Filkov, V. A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* **60**, 91–100 (2017).

[2] Gao, Z., Bird, C. & Barr, E. T. *To type or not to type: Quantifying detectable bugs in JavaScript*, 758–769 (IEEE, 2017).

[3] Kleinschmager, S., Hanenberg, S., Robbes, R. & Stefik, A. *Do static type systems improve the maintainability of software systems? an empirical study*, 153–162 (IEEE, 2012).

[4] Endrikat, S., Hanenberg, S., Robbes, R. & Stefik, A. *How do API documentation and static typing affect API usability?*, ICSE 2014, 632–642 (ACM, New York, NY, USA, 2014).

[5] Mayer, C., Hanenberg, S., Robbes, R., Tanter, E. & Stefik, A. Static type systems (sometimes) have a positive impact on the usability of undocumented software : An empirical evaluation (2012).

[6] Tambon, F. *et al.* Bugs in large language models generated code. *arXiv [cs.SE]* (2024).

[7] Zhong, L. & Wang, Z. Can ChatGPT replace StackOverflow? a study on robustness and reliability of large language model code generation. *arXiv [cs.CL]* (2023).

[8] Chong, C. J., Yao, Z. & Neamtiu, I. Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation.

arXiv [cs.CR] (2024).

[9] Ramírez, L. C., Limón, X., Sánchez-García, J. & Pérez-Arriaga, J. C. *State of the art of the security of code generated by LLMs: A systematic literature review*, 331–339 (IEEE, 2024).

[10] Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. *A history of haskell: being lazy with class*, HOPL III, 12–1–12–55 (ACM, New York, NY, USA, 2007).

[11] TypeScriptTeam. Type inference. https://www.typescriptlang.org/docs/handbook/type-inference.html. Accessed: 2024-2-19.

[12] Klabnik, S. & Nichols, C. The rust programming language. https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html. Accessed: 2024-2-19.

[13] Griesemer, R. & Taylor, I. L. An introduction to generics - the go programming language. https://go.dev/blog/intro-generics. Accessed: 2024-2-19.

[14] Tirronen, V., Uusi-Mäkelä, S. & Isomöttönen, V. Understanding beginners' mistakes with haskell. *J. Funct. Programming* **25**, e11 (2015).

[15] Chen, S. & Erwig, M. Counter-factual typing for debugging type errors. *ACM SIGPLAN Notices* **49**, 583–594 (2014).

[16] Heeren, B., Leijen, D. & van IJzendoorn, A. *Helium, for learning haskell*, 62–71 (ACM, New York, NY, USA, 2003).

[17] Zhang, D., Myers, A. C., Vytiniotis, D. & Peyton-Jones, S. Diagnosing type errors with class. *SIGPLAN Not.* **50**, 12–21 (2015).

[18] Lerner, B. S., Flower, M., Grossman, D. & Chambers, C. Searching for type-error messages. *SIGPLAN Not.* **42**, 425–434 (2007).

[19] Zhang, D., Myers, A. C., Vytiniotis, D. & Peyton-Jones, S. SHErrLoc: A static holistic error locator. *ACM Trans. Program. Lang. Syst.* **39**, 1–47 (2017).

[20] Fu, S. Goanna, a haskell type checker based on constraint logic. https://goanna.typecheck.me/ (2025). Accessed: 2025-4-12.

[21] Reis, C. L. I. Curry-howard correspondence.

[22] Pottier, F. & Rémy, D. 10 the essence of ML type inference. *cambium.inria.fr* .

[23] Hage, J. & Heeren, B. Strategies for solving constraints in program analysis (2006).

[24] Ignatiev, A., Narodytska, N., Asher, N. & Marques-Silva, J. On relating 'why?' and 'why not?' explanations. *arXiv [cs.LG]* (2020).

[25] Nelson, T., Danas, N., Dougherty, D. J. & Krishnamurthi, S. *The power of "why" and "why not": enriching scenario exploration with provenance* (ACM, 2017).

[26] Haack, C. & Wells, J. B. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming* **50**, 189–224 (2004).

[27] Stuckey, P. J., Sulzmann, M. & Wazny, J. *Interactive type debugging in haskell*, 72–83 (ACM, New York, NY, USA, 2003).

[28] Fruehwirth, T. Constraint handling rules - what else? *arXiv [cs.PL]* (2017).

[29] Junker, U. Preferred explanations and relaxations for over-constrained problems. *AAAI-2004* (2004).

[30] de la Banda, M. G., Stuckey, P. J. & Wazny, J. *Finding all minimal unsatisfiable subsets* (ACM, New York, NY, USA, 2003).

[31] Bailey, J. & Stuckey, P. J. *Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization*, 174–186 (Springer Berlin Heidelberg, 2005).

[32] Liffiton, M. H. & Sakallah, K. A. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Automat. Reason.* **40**, 1–33 (2008).

[33] Liffiton, M. H., Previti, A., Malik, A. & Marques-Silva, J. Fast, flexible MUS enumeration. *Constraints* **21**, 223–250 (2016).

[34] Zhao, W. & Liffiton, M. H. *Parallelizing partial MUS enumeration*, 464–471 (IEEE, San Jose, CA, USA, 2016).

[35] Bendík, J. & Černá, I. Biere, A. & Parker, D. (eds) *MUST: Minimal unsatisfiable subsets enumeration tool.* (eds Biere, A. & Parker, D.) *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 135–152 (Springer International Publishing, Cham, 2020).

[36] Chen, S. & Wu, B. Efficient counter-factual type error debugging. *Science of Computer Programming* **200**, 102544 (2020).

[37] Chen, S. & Noor, M. R. in *Improving type error reporting for type classes* Lecture notes in computer science, 19–38 (Springer International Publishing, Cham, 2022).

[38] Erwig, M. & Walkingshaw, E. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.* **21**, 1–27 (2011).

[39] Fu, S., Dwyer, T., Stuckey, P. J., Wain, J. & Linossier, J. *ChameleonIDE: Untangling type errors through interactive visualization and exploration* (arXiv, 2023).

[40] Zhao, E. *et al.* Total type error localization and recovery with holes. *Proc. ACM Program. Lang.* **8**, 2041–2068 (2024).

[41] Potter, H. & Omar, C. *Hazel tutor: Guiding novices through type-driven development strategies*, 10 (ACM, 2020).

[42] Robinson, J. 4 computational logic : The unification computation (2012).

[43] Fu, S. Goanna-IDE: A haskell IDE for type error debugging. https://goanna-ide.fly.dev/ (2023). Accessed: 2023-10-8.

[44] Simon Marlow. Haskell 2010 language report. Tech. Rep. (2010).

[45] Fu, S. Goanna's haskell language coverage. https://goanna.fly.dev/coverage.html (2023). Accessed: 2023-10-8.

[46] Wielemaker, J. & Costa, V. S. in *On the portability of prolog applications* Lecture notes in computer science, 69–83 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011).

[47] Fausak, T. 2022 state of haskell survey results. https://taylor.fausak.me/2022/11/18/haskell-survey-results/ (2022). Accessed: 2023-10-12.

[48] Charguéraud, A. Improving type error messages in OCaml. *Electronic Proceedings in Theoretical Computer Science* **198**, 80–97 (2015).

[49] Bhanuka, I., Parreaux, L., Binder, D. & Brachthäuser, J. I. Getting into the flow: Towards better type error messages for constraint-based type inference. *Proc. ACM Program. Lang.* **7**, 431–459 (2023).

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

[51] Seidel, E. L., Sibghat, H., Chaudhuri, K., Weimer, W. & Jhala, R. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proceedings of the ACM on Programming Languages* **1**, 1–27 (2017).

[52] Tsushima, K., Chitil, O. & Sharrad, J. *Type debugging with counter-factual type error messages using an existing type checker*, IFL '19, 1–12 (Association for Computing Machinery, New York, NY, USA, 2021).

[53] Wu, B., Campora, J. P., III & Chen, S. Learning user friendly type-error messages. *Proc. ACM Program. Lang.* **1**, 1–29 (2017).

[54] Chen, S., Erwig, M. & Smeltzer, K. Exploiting diversity in type checkers for better error messages. *J. Vis. Lang. Comput.* **39**, 10–21 (2017).

[55] Stuckey, P. J., Sulzmann, M. & Wazny, J. *Improving type error diagnosis* (ACM, New York, NY, USA, 2004).

[56] Pavlinovic, Z., King, T. & Wies, T. Finding minimum type error sources. *SIGPLAN Not.* **49**, 525–542 (2014).

[57] Wu, B. & Chen, S. How type errors were fixed and what students did? *Proceedings of the ACM on Programming Languages* **1**, 105:1–105:27 (2017).

[58] Németh, B., Choi, E., Makihara, E. & Iida, H. Investigating compilation errors of students learning haskell. *arXiv [cs.PL]* (2019).

[59] Stuckey, P. J., Sulzmann, M. & Wazny, J. The chameleon type debugger (tool demonstration). *arXiv [cs.PL]* (2003).

[60] Gamari, B. The glasgow haskell compiler. https://www.haskell.org/ghc/. Accessed: 2023-6-1.

[61] Hage, J. & Helium Contributors. Haskell helium - a haskell compiler (2023).

[62] Ferdowsi, K. The usability of advanced type systems: Rust as a case study (2023). 2301.02308[cs].

[63] McAdam, B. *Repairing type errors in functional programs.* Ph.D. thesis, University of Edinburgh (2002).

[64] Chen, S., Erwig, M. & Smeltzer, K. *Let's hear both sides: On combining type-error reporting tools*, 145–152 (IEEE, Melbourne, Australia, 2014).

[65] Previti, A. & Marques-Silva, J. Partial MUS enumeration. *Proc. Conf. AAAI Artif. Intell.* **27**, 818–825 (2013).

[66] Lee, Y., Jeong, S. & Kim, J. Improving LLM classification of logical errors by integrating error relationship into prompts. *arXiv [cs.AI]* (2024).

[67] Berglund, L. *et al.* The reversal curse: LLMs trained on "a is B" fail to learn "B is a". *arXiv [cs.CL]* (2023).