Shuai Fu

Abstract

Static typing is a fundamental feature embedded in the origins of programming languages and is recognized for its numerous advantages, such as bug prevention, improved code quality, and decreased maintenance costs. Nevertheless, it presents considerable challenges, including a steep learning curve and slowed development pace, leading to a preference in recent years for lightweight dynamic languages like Python and JavaScript over statically typed languages such as Java and C++. Paradoxically, while stronger and more advanced type systems prevent a greater number of bugs, they also make resolving type errors more difficult and pose a larger barrier for non-expert programmers. Over the years, significant efforts, including contributions of our own, have been made toward augmenting the usability of static types and type errors.

My research investigates the tasks of resolving type errors in Haskell, a function programming languages, and how these tasks can be improved. We propose a method of categorization of type errors by anlalyzing the patterns of the locations that lead to the type errors. This categorization reveals different challenges associated with different kinds of type errors. This understanding led to the development of three key interventions: ChameleonIDE, Goanna, and GeckoGraph. ChameleonIDE is a type error debugger that visualizes conflicting locations of type errors and provides a step-by-step interactive exploration of their occurrences. Goanna focuses on presenting all possible causes of a type error and, for each cause, offers a minimal yet comprehensive set of changes to resolve it. GeckoGraph, a graphical notation for types, offers a more concise language for representing type signatures in source code, documentation, and type errors. These contributions allow us to generalize the task of debugging type errors into several idiomatic interactions supported by common user interface design patterns. Building on this foundation, we envision a debugging system called *TypeTutor* that integrates these idioms to create a natural and user-friendly approach to type error resolution.

By adopting multiple human-centered design methods, our research identifies promising avenues for enhancing our interaction with static-type systems in modern programming environments. We hope these findings will inform future studies aimed at improving type error debugging and inspire the development of next-generation programming tools to overcome the challenges of resolving type errors.

![](data:application/pdf;base64,)

Doctoral Thesis

*Author:*

*Supervisors:*  
Professor Tim Dwyer  
Professor Peter James Stuckey

*A thesis submitted for the degree*  
  
  
[Faculty of Information Technology](http://abc.com)

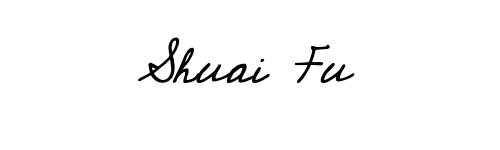
2024-12-24

© (2024)

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner’s permission.

I, , declare that this thesis titled, and the work presented in it are my own. I confirm that:

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, or any use of generative artificial intelligence technologies, except where due reference is made in the text of the thesis.

Signature: 

Print Name: Shuai Fu

Date: May 2024

This thesis uses content from the following publications. The co-authors of these works are primarily my supervisors, and the collaboration occurs mainly within my supervisory team. I fully acknowledge and deeply appreciate the contributions of all co-authors; without their input, the quality of this work would have been significantly diminished.

**Interactive Haskell Type Inference Exploration — Extended Abstract**

Authors: **Shuai Fu**, Tim Dwyer, Peter J. Stuckey

This extended abstract was published and presented at the International Conference on Functional Programming (ICFP) 2021 - Type Driven Development (TyDe) workshop. Although the thesis does not directly incorporate the content of this work, the research development closely follows the outline presented in this abstract. I contributed 70% of writing this abstract.

**ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration**

Authors: **Shuai Fu**, Tim Dwyer, Peter J. Stuckey, Jackson Wain, Jesse Linossier

This paper was published and presented at the International Conference on Program Comprehension (ICPC) 2023. Chapter [3](#chap:chameleon) incorporates content from this paper, with slight modifications to ensure coherence within this thesis. As the first author, I contributed 65% of the work, including development, experimentation, analysis, and writing the paper.

**Goanna: Resolving Haskell Type Errors With Minimal Correction Subsets**

Authors: **Shuai Fu**, Tim Dwyer, Peter J. Stuckey, John Grundy

When submitting this thesis, this paper was publicly available as a pre-print. Chapter [4](#chap:goanna) incorporates content from this paper, with slight modifications to ensure coherence within this thesis. As the first author, I contributed 65% of the work, including development, experimentation, analysis, and writing the paper.

**GeckoGraph: A Visual Language for Polymorphic Types**

Authors: **Shuai Fu**, Tim Dwyer, Peter J. Stuckey

When submitting this thesis, this paper was publicly available as a pre-print. Chapter [5](#chap:gecko-graph) incorporates content from this paper, with slight modifications to ensure coherence within this thesis. As the first author, I contributed 65% of the work, including development, experimentation, analysis, and writing the paper.

I owe a huge debt of gratitude to many individuals for their support and guidance. I extend my deepest thanks to my supervisors, Professor Tim Dwyer and Professor Peter J. Stuckey, for their scholarly mentorship and friendly banter throughout my candidature. I am immensely grateful to Professor John Grundy for his kindness and wealth of knowledge. I wish to express my thanks to my milestone panel members, Associate Professor Bernhard Jenny, Associate Professor Michael Wybrow, and Associate Professor Guido Tack, for informing my research with their expert insights. My gratitude goes to Julie Holden, Research Communication and Academic Language Specialist, for guiding me through the often complex terrain of academic writing. Most importantly, I owe a huge thank-you to my dear wife, Nina, for her unwavering financial and emotional support over the years.

All text and figures in this thesis are my own. I received editing advice from my supervisors and the university’s academic language service. Generative AI (OpenAI ChatGPT-4) was only used as a research tool to explore the prospects and limitations of large language models in programming assistance. The specific prompts and their context are detailed in Chapter [6](#chap:conclusion).

# Introduction

Programming languages are the media through which we communicate our ideas with machines, be it mathematical formulae or video games. Similar to human languages, programming languages consist of words, grammar, and meaning. Unlike conversing with humans, machines tolerate very little ambiguity and are not skillful at navigating confusion. Therefore, sometimes inconsistencies glossed over by humans can cause unexpected results in computer programs; a missing semicolon or extra empty space can lead to entirely different outcomes, sometimes resulting in disastrous failures. Throughout the history of programming, various ways were invented to ensure the program behaves as expected. Through this, a pattern emerges from our pursuit for program correctness: the more certainty a programmer attains through programming language rules and checks that their program will run correctly, the more difficulty the programmer has in conforming to those rules and checks. Further, when these rules and checks are violated, we meet the anguish of the machine in arcane jargon, sometimes all in uppercase letters.

In this thesis, we investigate the problem of type errors. Type errors are a subclass of program errors and are a common way for machines to signal that the program failed a “pre-drive” safety check. Although they are intended to guide programmers in writing better code, type errors are known to instill fear in beginner programmers and remain a persistent annoyance for even experienced developers. We identify what causes type errors, why they are hard to deal with, and why our existing tools fall short to help programmers combat type errors. We aim to transform the traditional, often terse, type error feedback into a dynamic and clear diagnostic system enhanced by interactive user interfaces.

In this chapter, we delve into two critical aspects of programming language design: their typing disciplines and the paradigms to which they subscribe. We discuss the nature of type errors, their characteristics, and the inherent complexities associated with improving them. Furthermore, we review the evolution of tools designed to aid programmers in deciphering and resolving program errors, addressing the notable gap between the tools to support type errors and runtime errors. Through these discussions, we seek to illuminate both the challenges and the ongoing efforts to enhance type error diagnostics.

## From types to program correctness

In programming language theory and practice, type systems are a widely adopted program validation method where types of expressions are checked against their usage. In the early history of programming, types were used to inform the compiler of how much memory needed to be allocated for each value. Now, type systems are much more powerful; programmers can express complex ideas, like communication protocols and concurrency characteristics. Conventionally, the discipline of typing is identified by the existence of a compile-time checking stage. A programming language is said to be **statically typed** if checks are performed before a program is executed. On the other hand, **dynamically typed** languages (colloquially called dynamic languages) will not complain about type mismatch prior to the program being run, and they do not facilitate describing the types (Fig. [1.1](#fig:typed-vs-untyped)). In dynamic languages, expressions like 4 + "2" may produce unintended consequences, for example, a runtime error causing the program to terminate early, or some languages may try to perform unexpected coercion of data types, producing the answer “42” (i.e. string concatenation instead of numeric addition).

Statically typed languages have a long history and are extremely popular, with examples among both the earliest of programming languages (such as FORTRAN and ALGOL) and the most widely used languages across all platforms (such as C  and its derivatives, including Java and C#). Static typing is also a core feature of the most advanced and renowned academic languages (such as ML and Haskell , as well as their derivatives, such as OCaML and Agda, respectively). In practice, statically typed programming languages offer many advantages. Type declarations and annotations add important contextual information about the expected use of variables and expressions. This additional context allows early error detection but also enhances code readability and promotes maintainability , especially in large collaborative projects. Explicitly written type annotation also creates opportunities to improve tooling support through intelligent compiler services and IDE (interactive development environment). Additionally, static typing enhances code documentation by clarifying contracts between library authors and users and, more generally, enhancing code reusability .

In comparison, dynamic typing has its own benefits and has an equal, if not stronger, influence in the computing world. In dynamically typed languages, a variable can hold different types of values because the type is checked during runtime. This often provides more flexibility than static typing since a function can be used to process different types of input values without any special semantics. However, this flexibility also carries risks, such as the example expression 4 + "2", where continuing execution with an incorrect value could have dire outcomes, especially when running in a production environment.

Languages within the Lisp family, such as Common Lisp and Scheme, epitomize the advantages of dynamic typing. In these languages, a prevalent design pattern involves functions accepting potentially different Lisp data types: atoms, lists, or symbolic expressions (S-expressions). The behavior of the function behavior changes based on the runtime inspection of these inputs. Additionally, dynamic languages often have a lower learning curve. Programmers are not burdened with declaring and adhering to strict type constraints, allowing them to focus more on immediate computational tasks. In the modern computing landscape, the popularity of dynamic languages like JavaScript and Python can be attributed to their flexibility and ease of use, which significantly lowers the barrier to entry for new programmers .

It comes as no surprise that large amounts of work were dedicated to deciding which of the two typing disciplines is superior. Over the years, numerous studies have attempted to compare these typing disciplines and their impact on various aspects of software development, including code quality , maintainability , error resolution , development speed for new features , and comprehensibility . Unfortunately, these studies have not led to a consensus, as the impacts of the typing disciplines can often be overshadowed by more dominant factors such as the style of the programming languages used and the skill levels of the programmers involved.

Despite the lack of universal agreement, conventional wisdom suggests that dynamically typed languages are often more suited for beginners and rapid prototyping, thanks to their flexibility and ease of use. Conversely, statically typed languages are typically favored in larger projects or environments where ongoing feature development and team collaboration are prevalent. This preference is due to the inherent type safety features, which help manage complexity and reduce certain errors as projects scale .

![](data:application/pdf;base64,)

Comparison of the same program dynamically typed in Python (Top) and with mypy—a static typing tool for Python (Bottom). The dynamic-typing rules of the Python interpreter apply, at run-time, the string concatenating version of the ‘+’ operator – regardless of the programmer’s intention. By contrast, mypy static-typing checks the type of the values passed to the plus function against the intention declared by the programmer through the type annotations on the function parameters (both int) and its return type (also int). The type check will fail until the programmer amends the function call with numeric arguments or otherwise explicitly converts the values from str to int.

## Functional Programming

Alongside static typing, **functional programming** represents another rigorous approach to software development, drawing inspiration from Alonzo Church’s lambda calculus in the 1930s . In functional programming languages, functions are the fundamental building blocks. They can be used as ordinary values, serving as both inputs and outputs for other functions, and can be combined to construct new functions. Programming in these languages centers around the composition of functions to perform high-level tasks.

A specific subset of functional programming languages, known as ‘pure’ functional programming languages, incorporates additional mathematical rigor through concepts such as immutable values and referential transparency. Immutable values mean all values, once declared, cannot be modified. Referential transparency ensures that functions do not perform tasks that depend on external systems, such as reading from a global state, connecting to network resources, or even printing to the console. These properties help mitigate undesirable programming behaviors similar to the way static typing does. For example, functional programming avoids issues like misaligned pointers and race conditions by disallowing mutable pointers and shared memory. Consequently, programs developed using these languages are more predictable and can be robustly tested and reasoned about . This significantly reduces the risks that external factors could unpredictably affect program behavior, such as network fluctuations or even the passing of time .

Functional programming is often advocated as an excellent choice for introductory programming courses due to its emphasis on mathematical reasoning and strict programming discipline . This discipline includes the clear separation of data from its transformations, fostering a structured approach to problem-solving.

|  |
| --- |
|  |

The figure compares two approaches to implementing the same function specification in different programming paradigms: imperative (top) and functional (bottom). The top section presents Python code, utilizing a loop and a mutable variable to accumulate sums. Conversely, the bottom section illustrates Haskell code, using a composition of two functions: sum and filter, the latter using a specified predicate in the form of an anonymous function. This exemplifies how imperative programming relies on the mutable state (the sum variable) to track progress, whereas functional programming leverages high-level abstractions and recursion to achieve similar outcomes.

Functional programming contrasts with other mainstream paradigms like object-oriented programming, which structures programs around objects combining data and behavior, and procedural programming, which focuses on a sequence of procedural steps. Despite the strengths of functional programming, object-oriented and procedural paradigms, exemplified by Java and C, respectively, remain more prevalent in commercial environments . These paradigms benefit from extensive legacy codebases, broad third-party library support, robust tooling, and familiarity. Lastly, the rigor in functional programming languages may raise the barrier to entry for beginners. For example, in many pure functional languages, printing to the terminal window, generally considered a basic technique to observe the execution of the program, exposes programmers to profound concepts like monad and side effects, which can be daunting to those who are new to the language.

## Statically Typed Functional Languages, The Best Of Both Worlds

Combining the disciplines described above, **statically-typed functional languages** employ both static typing and the principles of functional programming. The most common statically typed functional languages include Haskell, ML (with the OCaml dialect being the most popular among the family of ML languages), and F#. Of these, Haskell is the only “pure” functional language, with all variables being immutable by default. Descendents of Haskell, like Idris and Agda, include more advanced type-level features like dependent type and session type, allowing programmers to express extremely granular checks of potential software behavior before running the program. These languages often provide the strongest level of programming safety. It is often advertised that programs in these languages will be error-free if the source code passes the compiling stage, indicating that compilers are able to weed out a large number of programming errors. These safety properties allow statically typed functional languages to be used as proof assistants or formal verification tools. They prove the correctness (or incorrectness) of many systems, from web public key infrastructure to microcontrollers used in space programs .

Despite these safety benefits, the presence of these languages in the mainstream programming world remains underwhelming. This modest popularity is often attributed to high entry barriers, unfamiliarity with the paradigms, and advanced type systems that can be overwhelming for newcomers.

## Symptoms of Type Errors

The compiler is the medium through which programmers transform human intentions into machine instructions. When encountering errors, compilers often act like Oracles in ancient history, revealing to programmers obscure messages that can lead to confusion and the wrong course of action. Many studies have investigated the ineffectiveness of compiler error messages . Our exploration is centered around type error messages (a subclass of compiler error messages) in the Haskell programming language and its leading compiler, Glasgow Haskell Compiler (GHC). It is vital to mention that the challenges discussed here concerning error messages are common across compilers for various statically typed languages. Below, we delve into specific issues often observed in type error messages.

### Bias in Type Errors

A significant issue in type error messages is their inherent bias. Often, type errors can arise from multiple causes, but the error message might highlight only one. This issue is known as left-to-right bias in traditional type-checking algorithms, a notable shortcoming that limits the usability of error messages . For instance, as shown in Fig. [1.3](#fig:type-error-example), a programmer likely intended to add two numbers using the + operator instead of mistakenly using the concatenation operator ++. However, the type error reported by the compiler mistakenly points to the application of the integer literal 3, failing to suggest the more probable cause. We explore left-to-right bias further in Chapter [2](#chap:haskell-type-checking).

|  |
| --- |
|  |

The figure illustrates a programming error scenario featuring the source code with a type error (Top) and the corresponding compiler output (Bottom) using GHC 9.0.2. The error likely arises from the inappropriate use of the concatenation operator ++, rather than the addition operator +, on line 1, as inferred from the function name addTwoNumbers. Despite this, the compiler erroneously highlights line 3, where the function addTwoNumbers is applied to the integer 3, boldly assuming the correctness of the addTwoNumbers definition. This misdirection suggests a less likely source of error, potentially complicating the debugging process.

### Type Error Suggests Incomplete Cause

Often, type error messages do not fully present all the locations relevant to the type error. Addressing the error at the recommended location might not suffice to resolve the underlying problem. Consider the scenario in Fig. [1.4](#fig:type-error-example-2), where a function intended for list concatenation is erroneously applied to integer values. Here, the error message only reports the first argument 3, neglecting the second. This partial reporting leads to a situation where correcting the first argument to a list format, say [3], doesn’t rectify the error entirely; rather, it only updates to a slightly different error message.

|  |
| --- |
|  |

This figure depicts the same program as in Fig. [1.3](#fig:type-error-example), with one modification being the renaming of the function from addTwoNumbers to joinTwoLists. This change shifts the implied intent towards concatenating two List type values. The compiler still identifies an error at the same location—the integer literal 3—aligning more closely with the programmer’s understanding than in the previous example. However, the error message still misleadingly overlooks the erroneous type of the second argument, 4, as it, too, is an invalid input for a list concatenation operation. Therefore, modifying only the first argument is insufficient to correct the type error.

### Missing Links In Type Errors

One of the most frustrating aspects of type errors is that they do not show the complete pathway of how the compiler decided on the type error. In the example in Fig [1.5](#fig:type-error-example-3), the programmer may intend to compare to a char literal ’ ’ instead of string literal " " on line 1 in the function isSpace. However, multiple clues contribute to this conclusion. The definition of the function trimWhiteSpace, the application of filter isSpace a, the definition of isSpace, and even the type signature of filter all contribute to the logical reasoning. However, none of these are included in the actual error message.

|  |
| --- |
|  |

In Haskell, the String type is an alias for [Char], meaning the two types are equivalent. The function trimWhiteSpace, as used on line 3, processes a String (or list of Char values) to filter out whitespace characters. However, the function’s filter condition, isSpace, erroneously compares two String values instead of Char types. This is likely a type intending the char literal ’ ’ rather than the string literal " ". While the type error message correctly identifies the mismatch, it fails to provide crucial contextual clues that would obscure the nature of this error.

### The Use of Obscure Language

Error messages are frequently plagued by technical jargon and convoluted phrasing, which can be particularly off-putting for novices. An example is the error message, No instance for (Num a) arising from the literal ‘3’, which is an inept way of suggesting a type mismatch involving character and numeric types. In fact, this is often the common behavior across many programming languages and has been shown in many studies . Thus, concerted efforts show that rephrasing the error message to be clearer and more structured positively affects programmers’ ability to solve these errors.

## The Challenges Of Making Good Type Errors

After introducing some typical symptoms of unhelpful type error messages, we now explore some fundamental challenges to improve type error messages.

### Types Are Complex

The advantages of statically typed languages derive from their robust type-checking systems, which, ironically, also introduce significant complexities. Language and tool designers frequently overlook the intrinsic complexity of type systems. Type checking in these languages can exhibit Turing-complete characteristics, which might lead to non-terminating processes . TypeScript introduces ‘conditional types’, a feature that allows sophisticated computations at the type level (Fig. [1.6](#fig:ts-conditional)), posing challenges in explaining errors when these type-level computations fail due to a lack of adequate debugging tools.

|  |
| --- |
|  |

TypeScript includes a type-level feature known as ‘conditional types,’ which enables the declaring of a type based on the outcome of a condition check. This capability is sufficient to construct a Turing-complete language within the TypeScript type system. The figure illustrates a simple language that supports booleans, numbers (using Church encoding), and functions that operate on these values. Since this implementation is implemented entirely at the type level, it does not generate any executable code for runtime evaluation, functioning purely within the compile-time environment. The recreational value aside, these type-level features often lead to elaborated “essay of types”, hindering understanding and usability. This example was run with TypeScript 5.7.2.

### Clues For Type Checking May Be Implicit

The task of understanding how types are assigned in a program grows considerably more complex when the programming language employs global type inference. A programming language that uses global type inference allows programmers to forgo the task of writing type annotations. Instead, the type checker automatically deduces the appropriate types for each expression based on the context. Although it succeeded in providing rigorous type checking without the hassle of manually writing type annotations, type inference has been shown to cause many usability issues for its implicitness.

Even in programming languages that do not utilize type inference, certain typing rules remain opaque to programmers. For instance, rules on permissible values for equality check using the == operator vary across programming languages; some languages disallow comparison of lists, and others are fine with lists but disallow comparison of floating-point numbers. In languages that support a record data type, programmers must navigate additional complexities, such as whether adding or removing a field from a record maintains type correctness (Fig. [1.7](#fig:row-polymophism)). This often involves an understanding of nuanced language-specific rules, including concepts like covariance, contravariance, and subsumption.

![](data:application/pdf;base64,)

Row polymorphism is a common type-level feature, allowing record values with optional more fields to be a subtype of the record without these fields. This enhances flexibility in structuring data collections but also introduces complexity in understanding the relations between types. The figure illustrates an example where programmers try to apply the record values alice and bob to the function greet. Because of the rules of row polymorphism and the fact that function arguments are covariant, only one value constitutes a valid input. This example was run with TypeScript 5.7.2.

These challenges amplify the difficulty in designing intuitive type error messages. The complexity of type systems demands that designers have a profound understanding of both theoretical and practical aspects of language implementation—knowledge often possessed only by the core language developers. Moreover, the implicit nature of modern programming languages requires that type errors be designed on a case-by-case basis, a daunting task given the limited number of contributors who possess the necessary expertise.

## The Lack of Type Debugging Tools

Debugging programming errors has been an unpleasant but crucial part of programming since the inception of computing, tracing back to as early as 1949 . However, the tools and support for debugging type errors have not evolved significantly and do not match the advancement of those available for runtime errors.

For runtime errors, one of the simplest and most effective debugging techniques is to insert print statements. As Brian W. Kernighan, one of the creators of the Unix system, once noted, “The most effective debugging tool is careful thought, coupled with judiciously placed print statements” . Furthermore, breakpoint debugging has become a staple in nearly all programming Integrated Development Environments (IDEs), allowing for intermittent code execution and inspection (Fig. [1.8](#fig:breakpoint)). Research in error debugging tools continues to propose novel ideas. For instance, ZStep94 enhances traditional breakpoint debugging by eliminating the need to set breakpoints, thus enabling programmers to view and navigate through the historical values that expressions take throughout execution (Fig. [1.9](#fig:zstep94)). Another innovative tool, WhyLine , aligns with the principles of a natural programming environment by allowing programmers to ask “why” questions and “why not” questions about certain program behavior.

Despite these advancements in runtime debugging tools, the development of type debugging tools seems to have stagnated. Most programming languages and development environments still handle type errors in ways reminiscent of early languages like FORTRAN and ALGOL. In recent years, some modern programming languages like Elm and Rust have made efforts to improve type error reporting, but these enhancements are often superficial and do not fundamentally change the debugging experience .

|  |
| --- |
|  |

The figure shows the debugging of a Python program using a breakpoint and expression evaluation in PyCharm, a widely used interactive programming environment (IDE) for Python. Program execution is paused at the breakpoint set on line 4, and with each pause, the user-defined expression i \* j is re-evaluated. This common debugging interface is available in nearly all modern programming editors and integrated development environments.

|  |
| --- |
|  |

Debugging a Lisp program by inspecting all the historical values of an expression, navigating through execution history, and reviewing live visualization replay. All these features are provided in ZStep94.

## Research aim and objectives

This research is driven by the acknowledged difficulties in debugging type errors and the lack of significant advancement in the tools and techniques used for presenting and resolving these errors. **We aim to improve both the usability and accessibility of type error diagnostics, making statically typed languages more approachable for developers at all levels of expertise**. To achieve this, we have established two main objectives in order to tackle the inherent complexity of type errors and provide the support tools to manage them.

### Objective 1. Provide programmers with the comprehensive knowledge needed to understand and resolve type errors.

Current representations of type errors in most compiler tools are often insufficient for straightforward resolution without further investigation by the programmer. They typically provide only the location of the type check failure, the expected type, and the actual type encountered. This conventional approach, as discussed in the previous sections ([1.4](#sec:symptoms)), lacks practicality and clarity. We intend to address the need for a richer, more informative explanation of type errors by evaluating the limitations of existing systems and analyzing how programmers tackle type errors in practice.

#### Objective 1.1 To Encompass Multiple Potential Causes Of A Type Error

An essential aspect of this research is to challenge and expand beyond the bias found in traditional type error reporting (Section [[subsec:bias]](#subsec:bias)). It is crucial to inform programmers that multiple potential causes and resolutions are almost certain to be present in every type error. A key goal is to communicate the dimension of multiple potential causes effectively. In addition, for each potential cause, programmers need a clear explanation of where the offending code is, which typing rules are violated, and, most importantly, how the meaning of the program will change after the error is resolved.

#### Objective 1.2 To Accurately Report Relevant Locations Contributing to Type Errors.

Current tools often pinpoint a single location for a type error, a method that has attracted considerable criticism for its inefficiency. Programmers frequently need to scan beyond the initially reported location. We propose to enhance type error reporting by identifying and detailing all relevant error-contributing locations across the codebase.

#### Objective 1.3 To Give Reasons And Support Human Understanding

Understanding type errors goes beyond pinpointing the location in the code. Internally, type errors can be caused by mismatched types, unmatched type class constraints, and trying to construct infinite types, among other reasons. Externally, type errors can be caused by typos, outdated type annotation, incomplete implementation, etc. Our goal is not only to locate these errors but to explain them in a way that logically supports the programmer’s understanding and troubleshooting process, covering both internal causes and external causes.

### Objective 2. Support Programmers To Type Errors Through Interactive Modern Programming Environments.

We focus on integrating comprehensive type error encoding within an interactive programming environment to streamline workflows in statically typed languages. Although objective 1 focuses on enriching the information of type error, we acknowledge that an increase in information does not necessarily correlate with enhanced understanding. Thus, we also attempt to present the details of type errors intelligently to optimally support comprehension and resolution.

#### Objective 2.1 Visualize The Key Concepts Of Type Errors Effectively

Traditional text-based error reports can be difficult to navigate and understand. We explore innovative methods to present what used to be displayed as plain text (type signatures, locations in source code, call stacks, dependency graphs) in graphic user interfaces. Examples include in-line highlighting within the code editor and graph-based reasoning steps.

#### Objective 2.2 Use Interactive Tools for Investigating and Resolving Type Errors.

This objective focuses on using interactive techniques to improve the usability of type errors. This includes using interface elements that allow user input (hover, click, drag and drop, etc.) to trigger common debugging tasks. In addition, we attempt to provide context-sensitive debugging aids that adjust the interactive response based on the current performing task, error type, and individual user preferences.

## Contributions

This thesis presents four key areas of contribution. First, it introduces a categorization of type errors, which has informed the development of two distinct systems: ChameleonIDE and Goanna. Each system is tailored to address specific challenges within one or more categories of type errors. Finally, we introduce GeckoGraph, a visual language designed to enhance the understanding and use of polymorphic types in programming.

### A categorization of type errors

To address the complexities in explaining type errors comprehensively, we have classified type errors into three categories based on how human programmers perceive type errors. Formal definitions and a detailed discussion of these categories appear in Chapter [2](#chap:haskell-type-checking). It is crucial to understand that these categories are not mutually exclusive; for example, a multi-witness type error may also be a multi-step type error.

A **multi-step type error** involves a chain of logical inferences steps based on the available evidence of the error. Fig. [1.10](#fig:multi-step-example)) illustrates the chains of inference in the assignment of a (Line 1), the equivalence of a (Line 1 and Line 2), the assignment of b (Line 2), and the equivalence of b (Line 2 and Line 3). Removing anyone from the chains will resolve the conflict. Understanding and communicating the interconnections within these chains is crucial to resolving multi-step type errors effectively.

|  |
| --- |
|  |

This illustration depicts a multi-step type error in Haskell. In the program (left panel), the string literal "True" is assigned to the variable a on line 1, followed by the assignment of a’s value to variable b on line 2. Subsequently, on line 3, the program attempts to use b as a boolean condition in an if expression. The error analysis can be visualized as a "chain" of reasoning, represented by the continuous line (right panel), tracing the propagation of the type mismatch through the sequence of assignments and usage.

A **multi-witness type error** involves multiple pieces of evidence supporting the same potential type assignments. For instance, as shown in Fig. [1.11](#fig:multi-witness-example), multiple pieces of evidence (lines 3,4,5) support that a has the type Int -> String. On the other hand, a single piece of evidence (line 2) shows that a has the type Int -> Char. Clarifying and juxtaposing this discrepancy in witnesses is key to supporting understanding this type of error.

|  |
| --- |
|  |

This figure presents the analysis of a multi-witness type error in Haskell. In the source code (Left), a type conflict arises from the case expression, which could be interpreted as Char based on the value in line 2, or as a String based on the values in lines 3, 4, and 5. The type conflict is graphically illustrated (Right) with String values marked in pink and the Char value in blue. The majority of String value witnesses suggest the lone Char literal ’I’ on line 2 might be a typographical error. This discrepancy underscores the significance of witness count in identifying the likely error source.

A **multi-party type error** involves several potential type assignments, each supported by distinct evidence. Fig. [1.12](#fig:multi-party-example) presents a typical multi-step type error: the expression d can not be assigned a type because 3 pieces of evidence on line 1 suggest 3 potential types for d. In practice, addressing such errors requires breaking them down into multiple type errors of simpler forms.

|  |
| --- |
|  |

This figure analyzes a multi-party type error in the source code (Left), where the list d could potentially be typed as [Int], [String], or [Char]. The disagreement is visually represented (Right) by three distinct colors, each corresponding to the cause of one of the possible types. This scenario differs from previous examples as resolving the type error requires adjustments at multiple locations within the code, not just a single change.

### Explain Multi-Step Type Errors Through Chain-Of-Thought Visualization

We contribute **ChameleonIDE**, an interactive Haskell debugging tool that identifies all relevant error-contributing locations and illustrates the logical relationships between these locations.

Our contribution in ChameleonIDE benefits from the existing work on Chameleon . Originally, Chameleon was developed as a command-line tool. Its novel idea of interactive debugging allows programmers to view different possible type error explanations and type assignments by issuing console commands. In addition, Chameleon successfully encoded the Haskell type system, with the addition of functional dependency, entirely in constraint handling rules . ChameleonIDE shares the theoretical foundation with the original Chameleon, providing a modern implementation with many advanced features. Notably, ChameleonIDE provides a novel debugging interface to interactively explore the chain of reasoning in a multiple-step type error, allowing programmers to interactively develop a panorama view of the type error. In addition, ChameleonIDE uses an adaptive interface design; programmers can switch between different granularity of information based on their personal preferences. ChameleonIDE is evaluated in a series of user studies with Haskell programmers.

### Resolve multi-witness and multi-party errors with Minimal Correction Subsets

We provide **Goanna**, a Haskell type error debugging tool. Goanna enhances type error debugging in Haskell by identifying all relevant error-contributing locations and presenting all possible causes and their respective resolutions. It ranks the potential causes of type errors based on their likelihood, helping programmers identify the root causes with minimal interactions with the tool. Goanna is evaluated by benchmarking its accuracy, conciseness, and performance against the state-of-art type-checking tools.

### Visualizing Polymorphic Types

We contribute **GeckoGraph**, a graphic notation for Haskell types. GeckoGraph encodes the same information as type signatures but uses colors, shapes, and symbols to make certain structures easy to identify at a glance. GeckoGraph uses visual elements to improve the understanding of complex type concepts, such as type classes and qualified constraints. GeckoGraph offers clarity when reading complex type signatures and comparing multiple types. The effectiveness of using GeckoGraph in programming is evaluated in a large-scale user study, consisting of programmers of all experience levels.

## Thesis outline

This thesis is structured into two main parts. The initial chapters (Chapters [1](#chap:introduction), [2](#chap:haskell-type-checking)) build a comprehensive foundation on type systems and programming languages, setting the stage for our research contribution in subsequent chapters. The latter chapters focus on distinct contributions that tackle various challenges in type error debugging.

### Chapter 1: Introduction

Chapter 1 sets the stage by discussing the fundamental concepts of programming languages and type systems. It explores the trade-off between enhancing program safety and optimizing usability, a central dilemma in programming language research and software engineering practice. The chapter highlights common issues and technical challenges associated with type errors, establishing the motivation for this research. It also outlines the specific aim and contributions of the study, providing a roadmap for the thesis.

### Chapter 2: Haskell Type Checking

This chapter delves into the fundamental concepts underlying type systems and programming languages. It outlines traditional methods employed in Haskell type checking, error slicing, and interactive debugging. Additionally, this chapter discusses tools and techniques used in constraint satisfiability analysis that are integral to later developments in the thesis. The categorization of type errors is revisited and redefined, building on the definitions introduced in this chapter.

### Chapter 3: ChameleonIDE – Interactive Type Error Visualization and Exploration

Chapter 3 presents ChameleonIDE, a system designed to enhance the debugging of type errors through interactive visualizations. It begins with a discussion on the typical pitfalls of existing error messages and outlines the motivation behind ChameleonIDE. We then detail the system’s design, features, and development process, including iterative prototyping. A series of empirical studies were conducted to assess ChameleonIDE’s effectiveness compared to traditional compiler tools.

### Chapter 4: Goanna – Finding All Type Errors Using Minimal Correction Subsets

Chapter 4 introduces Goanna, a Haskell type error debugging tool that incorporates novel features such as suggesting fixes for type errors and cross-module debugging capabilities. It starts with highlighting the limitations of conventional compiler error messages and progresses to describe Goanna’s capabilities in identifying all causes and ranking potential causes by their likelihood. The implementation tactics and heuristic methods used in Goanna are discussed, followed by an empirical evaluation of the system’s accuracy, conciseness, and performance based on real-world Haskell code examples.

### Chapter 5: GeckoGraph - A Graphic Notation for Haskell Types

Chapter 5 introduces our design of GeckoGraph. GechoGraph is a graphic notation for Haskell types. This chapter starts by underscoring the challenges of understanding and using polymorphic types. It then introduces GeckoGraph, how it addresses these challenges, and how to constructively transform text-based type signatures to GeckoGraph. We then show our experiment on evaluating the effect of using GeckoGraph to help solve program tasks. Lastly, we discussed the strengths, limitations, and potential applications of GeckoGraph and type visualizations in general.

### Chapter 6: Conclusion

The final chapter reiterates the contributions of the thesis. It discusses potential avenues for future tool development and research opportunities, forecasting the future landscape of programming language research. The chapter concludes with reflections on the importance of improving type error diagnostics, and encouraging ongoing investigation and innovation.

# Haskell Type Checking

In Chapter [1](#chap:introduction), we briefly explored the history of programming languages with an emphasis on their practices in type-checking and the paradigms they employ. This chapter focuses closely on Haskell, outlining why it is particularly well-suited for studying improvements in type error detection. We delve into the common techniques for performing type-checking on Haskell programs, specifically the Hindley-Milner type inference system and Algorithm W.

Furthermore, this chapter examines various approaches to providing better error messages, including Algorithm M, type error slicing, and interactive debugging. We also discuss key developments in the field of constraint satisfiability, highlighting their significance in enhancing type error messages. Finally, we revisit the categorization of type errors, this time in light of the aforementioned techniques and tools.

## Why Haskell

Haskell, introduced in 1990, is a functional programming language renowned for its enforcement of pure functional principles, lazy evaluation, and an expressive type system . The primary focus of our work is to develop type debugging systems for Haskell that leverage this expressive type system and address its limitations. Although we plan to extend our work to other languages, there are compelling reasons for initially focusing on Haskell.

### Haskell In Education

As discussed in Chapter [1](#chap:introduction), Haskell plays a significant role in undergraduate computer science education and was designed with this educational purpose in mind. Many textbooks targeting first-year computer science courses and introductions to functional programming recommend Haskell as a pedagogical tool . Proponents argue that Haskell serves as an ideal platform for teaching programming and rigorous reasoning due to its closeness to mathematical objects. Students often find that Haskell "elegantly admits solutions that are difficult to formulate in imperative languages (minor editing)," a sentiment echoed by computer scientist Edsger W. Dijkstra in a letter advocating for the use of Haskell for freshmen at the University of Texas instead of Java.

Despite the enthusiasm of its advocates, Haskell’s actual adoption in universities has been modest. One frequent challenge in teaching Haskell is helping students overcome programming errors . Some recommend using a simplified subset of Haskell’s features to avoid the frustrations associated with complex type errors . Our research is motivated by both the desire to enhance Haskell’s educational value and the real-world challenges associated with teaching the language.

### Haskell’s special position in programming language research

Although the full semantics of Haskell have never been formally defined , this has not deterred its use as a platform for programming language innovation or for meaningful academic contributions. Subsets of Haskell can be formalized for rigorous language study . Haskell’s dual presence in academia and the mainstream programming world uniquely positions it as a testing ground for new programming language features and ideas. Indeed, features such as Haskell’s list comprehension have been adopted by languages such as Python and Julia, and its generic type systems have influenced languages such as Java, C#, and Rust.

Our research in Haskell benefits from the language’s relatively small and well-studied specification, enabling rapid implementation of new ideas and swift feedback. We also enjoy support from Haskell’s welcoming and active community, which helps guide our efforts towards improved correctness and usability.

## Hindley-Milner Type Inference and Algorithm W

Type inference, also known as implicit typing, is a technique to reduce the number of occurrences of manually ascribing types. Almost all languages today employ a certain level of type inference. For instance, Java versions prior to 10 required users to explicitly declare variable types, which often led to verbosity and redundancy, as shown in Fig. [2.1](#fig:example-java). Java later introduced the var keyword (from version 10 onwards), allowing the type of a variable to be inferred at compile-time based on the assigned value .

|  |
| --- |
|  |

An example of a typical Java program with and without using the var keyword. Before (Top), programmers have to annotate the type identical to the value initializer. After Java 10, this is solved using the local type inference with the var keyword (Bottom).

In language like Haskell and ML, not only is type inference applied, but its power to detect type error is hugely amplified by the use of the Hindley-Milner type inference . The Hindley-Milner Type System, named after its inventors Roger Hindley and Robin Milner, is a type system that can automatically infer the types of expressions in a language with no annotations required. It is a foundational part of the type system in many functional programming languages, including ML, Haskell, and Elm. The system provides polymorphic typing, meaning that a variable can be assigned multiple different types automatically based on its usage context, making it easier to write flexible, reusable code without compromising type safety. In many languages that use this system, it is proven that every expression will be assigned a most general type (principal type) based on its usage.

In simpler type inference systems, such as the one used in the Java example (Fig. [2.1](#fig:example-java)), type inference is local; it occurs within individual assignments, and the compiler may raise errors if it cannot infer types from a single statement. In contrast, the Hindley-Milner system employs a more global approach. It defers type constraint resolution and continues to assess the rest of the program, thereby increasing flexibility and reducing the immediateness of type errors.

|  |
| --- |
|  |

Illustration of Hindley-Milner type checking on a lambda expression. Initially (Step 1), fresh type variables and are introduced to assign types to and , with serving as a placeholder for the expression’s right-hand side. It then examines the right-hand side ’if’ expression, creating three additional variables: for the condition, and and for the left and right branches, respectively. Unification constraints are then applied, stipulating as a boolean , and requiring and to match . The process continues traversing the syntax tree until Step 8, where a type mismatch occurs as , previously unified with an integer type, is attempted to be unified with a boolean type, leading to a type inference error.

This sophisticated type inference mechanism found in the Hindley-Milner system underpins its wide adoption in the realm of functional programming. It exemplifies a shift towards more intelligent compilers that enhance developer productivity and code maintainability by abstracting the complexities of explicit type management.

**Algorithm W** The foundational mechanism of the Hindley-Milner type system is Algorithm W, a method to deduce the most general types for each expression in programming languages that support polymorphic types. This algorithm starts by assigning a unique type variable to each expression. It then recursively analyzes the structure of the program, generating a set of type constraints by inspecting each syntax node—as shown in Fig. [2.2](#fig:hindley-milner). This process involves the creation of new type variables and the unification of these variables with others or with specific types.

Unification is a critical step: It attempts to make different type variables agree by finding a common type that satisfies all constraints. If unification is unsuccessful at any point, the type-checking process halts immediately, indicating a type error. The algorithm reports the expression where the last constraint was produced, but this is often reported as the sole source of the error.

Despite the seismic influence of type theory and the conciseness of the programming style it helps achieve, the Hindley-Milner type system can be challenging in terms of usability. One notable drawback is how it handles error reporting. When type inference fails, the error messages can be cryptic and difficult to interpret. This is because the algorithm typically reports only the final constraint that led to the failure, omitting earlier but potentially relevant constraints. Unlike simpler cases in languages like Java, where a type error might be isolated to a single line, in a Hindley-Milner-based system, the underlying issue could be located anywhere in the program. This often leaves programmers scanning extensive sections of code to identify the root cause of the error.

This aspect of Algorithm W highlights a trade-off between the power of the type inference offered by the Hindley-Milner system and usability, notably in terms of error diagnostics and clarity. This has prompted ongoing research into improving error reporting in systems based on this algorithm, seeking to make them more accessible and easier to use for programmers.

**Algorithm M** In response to the usability challenges posed by Algorithm W, an alternative approach known as Algorithm M was proposed . This method modifies the direction of type inference from a bottom-up to a top-down process. By carrying constraints from the outer structure of the program to the substructures, Algorithm M allows for more contextual analysis, resulting in more intuitive error messages in certain scenarios, as illustrated in Fig. [2.3](#fig:algorithm-m-1).

![](data:application/pdf;base64,)

Comparison of error reporting between Algorithms M and W. In this example, Algorithm M identifies a more precise and plausible error location, whereas Algorithm W only provides a general, less specific error location.

|  |
| --- |
|  |

Analysis of error detection where Algorithm M fails to identify the correct error location, suggesting that ’-’ is causing the error. Despite this, Algorithm W continues to provide a general, less specific error location.

Despite its improvements in some cases, Algorithm M does not consistently outperform Algorithm W in terms of error clarity and can still produce misleading results, as shown in Fig. [2.4](#fig:algorithm-m-2). Both algorithms essentially face the same fundamental limitation: type inference is conducted step-by-step through unification, and once constraints are unified, they are discarded and unretrievable. Consequently, errors detected by these algorithms may not accurately reflect the root issues if multiple possible causes of types exist.

A lesson that can be learned from both Algorithm W and M is the recognition that no inference method ensures that it will perfectly align with the programmer’s intentions. Rather than ambitiously pinpointing where the root cause may lie, a more realistic goal might be to represent type errors succinctly without making assumptions.

## Type Error Slicing

Type error slicing is an advanced technique intending to enhance the meaningfulness of type error messages provided to programmers. This approach, referenced in works by Tip and Dinesh  and Haack and Wells , improves upon traditional type inference methods by postponing the unification process until all relevant constraints have been established. Type error slicing incorporates two principal innovations:

1. **Labeled Constraints:** This involves assigning identifiable labels to constraints based on their locations within the program. These labels are subsequently used to trace and diagnose parts of the code associated with a type error, offering clearer insights into the origins of an error.
2. **Minimal unsatisfiability analysis:** This technique focuses on identifying the smallest set of conflicting constraints that cannot be satisfied simultaneously. This means finding the necessary evidence needed to reproduce the failed unification, but nothing more than that. This approach significantly improves the diagnosis of type errors by avoiding the biases found in both Algorithms W and M.

![](data:application/pdf;base64,)

Illustration of type error slicing applied to the Haskell function fac. The highlighted "slices" indicate conflicting type inferences for the variable n: it is inferred as Char since it is compared against ’0’, yet it must be an integer because it is used in arithmetic subtraction. This conflict leads to the identification of a type error.

In contrast to conventional methods, type error slicing provides a comprehensive view of error locations, as demonstrated in Fig. [2.5](#fig:type-error-slicing). It allows programmers to understand type errors more thoroughly by presenting both sides of typing conflicts. It has less chance of missing critical clues, thus facilitating a more informed debugging process.

Over time, type error slicing has become a preferred method for optimizing type errors in research. Innovations in this area include more efficient methods of finding minimal unsatisfiable subsets . Type error slicing has also been enriched by exploring various underlying constraint languages to encode more complex type system features, such as those using SMT (Satisfiability Modulo Theories) and Constraint Handling Rules . These advancements have broadened the scope and applicability of type error slicing, making it a valuable tool for programming language development and error diagnosis.

## Interactive Type Debugging

Traditional type error slicing methods have been instrumental in localizing type errors by highlighting a comprehensive yet minimal set of related code locations. Despite the advantages, there are several key limitations associated with the localization and diagnosis in type error slicing:

|  |
| --- |
|  |

In this case, type error slicing is less useful, highlighting every possible location in the source code.

1. **Limited Code Reduction:** Although MUS-based error slicing can effectively narrow down the potentially defective code areas, studies such as Binkley’s empirical analysis on program slicing suggest that this method only reduces about 30% of the code that needs to be examined to understand a type error. The minimal nature of MUS often prevents further reduction, which can be insufficient for complex errors, as demonstrated in Fig. [2.6](#fig:slicing-counter-example), where virtually every location in the program could be implicated in the error.
2. **Lack Of Enhanced Information Encoding:** The existing MUS-based approaches might highlight critical error sites, but they often fail to capture the breadth of information required to fully understand and resolve type errors, especially in systems with complex type features.

To address these limitations, pioneering tools such as Chameleon have been developed. Initially introduced in the early 2000s, Chameleon served as a command-line tool aimed at improving type error reporting within the Haskell programming language. Unlike typical type-error slicing methods, Chameleon utilizes a more expressive constraint language, the constraint handling rules (CHR), which enables the support of more flexible relational constraints tailored to advanced type-level features such as type classes and functional dependencies.

A key innovation of Chameleon is its adoption of interactive type debugging. This approach not only identifies potential error locations, but also reveals that there could be multiple sets of potential types, most general unifiers, that could resolve the type issues present in the erroneous program. These types can be traced to distinct sets of code locations.

With interactive type debugging, programmers can query the type information of any possible resolution, effectively allowing programmers to ask counterfactual questions such as “what would the type of the expression be if the type error is fixed". This interactive process significantly advances the programmers’ understanding of type errors by allowing them to explore different resolution paths and better comprehend the type system’s behaviors and expectations.

Interactive type debugging thus represents a significant evolution in the handling of type errors, offering a more dynamic and informative approach compared to static slicing methods. This development underscores a shift towards more user-centered diagnostic tools in programming, where clarity and interactivity play vital roles in problem-solving.

|  |
| --- |
|  |

Demonstration of the interactive debugging feature in Chameleon. The source code is highlighted in two distinct styles (solid and outline), suggesting two potential types assignable to the variable w (left). Complementary details about these types are further explored through a command line query interface (right).

## The Analysis of an Unsatisfiable System

A slicing-based approach benefits significantly from the established tools and techniques in constraint satisfiability. Key instruments in this domain include the Minimal Unsatisfiable Subset (MUS), the Minimal Correction Subset (MCS) and the Maximal Satisfiable Subset (MSS).

### Minimal Unsatisfiable Subset

The MUS is a widely used tool in programming error analysis; it is used in various programming slicing tools . The MUS represents the smallest possible subset of a constraint system that still remains unsatisfiable, meaning that no values can be assigned to the logical variables without causing a conflict. This subset is crucial in determining the minimal set of code locations responsible for a type error.

Formally, a minimal unsatisfiable subset (MUS) of a constraint system is a subset such that is unsatisfiable and 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 not" question , in other words, finding the minimal contributors that prevent the system from emitting a certain behavior (in our case, such behavior is to become well-typed).

It is important to note that an unsatisfiable constraint system may contain multiple MUSes. For instance, consider a system with 5 propositional constraints depicted in Fig. [2.8](#fig:mus-example) - the system is infeasible as a whole. Within it, three MUSes can be identified. Each MUS is minimal, meaning removing any single constraint results in a satisfiable system. Typically, an incremental algorithm is employed to discover a single MUS. This algorithm starts from an empty set and adds one constraint at a time until the set is no longer satisfiable, indicating that the last added constraint must be an element of the MUS.

|  |
| --- |
|  |

An example of a constraint system modeled in propositional logic featuring 7 constraints and 2 logic variables a and b. The system is initially infeasible. Three potential Minimal Unsatisfiable Subsets (MUSes) can be identified. Each MUS is itself unsatisfiable. More critically, the removal of any single constraint from an MUS yields a satisfiable set of constraints.

### Minimal Correction Subset and Maximal Satisfiable Subset

Beyond the MUS, the Minimal Correction Subset (MCS) represents the smallest group of constraints that, when removed, resolves the infeasibility (defect) of the system. Formally, a minimal correction set (MCS) of a constraint system is a subset such that is satisfiable and is unsatisfiable. This "correction" subset is crucial for identifying the minimal changes needed to rectify errors in a program. Thus, MCS is often associated with addressing the “Why" question , in other words, finding the minimal reasons that trigger a certain behavior. In the example (Fig. [2.9](#fig:mcs-example)), 4 MCSes can be found. Removing each of these will result in a satisfiable set of constraints. Note that the satisfiability of MCSes themselves is not important.

|  |
| --- |
|  |

Depiction of a constraint system in propositional logic, consisting of 7 constraints and 2 logic variables a and b. Initially, this system is infeasible. Four Minimal Satisfiable Subsets (MCSes) can be identified, where the removal of each individual MCS from the constraint system leads to a maximally feasible system.

The Maximal Satisfiable Subset (MSS) is the complement of an MCS. A maximal satisfiable subset (MSS) of a constraint system is a subset such that M is satisfiable and is unsatisfiable. In practical terms, an MSS represents the largest portion of the constraint system that does not contribute to the type error, offering insights into the potential outcome of a working system after removing the defect. An example of MSSes in a constraint system can be seen in Fig. [2.10](#fig:mss-example).

|  |
| --- |
|  |

Depiction of a constraint system in propositional logic with 7 constraints and 2 logic variables a and b. While the full system is infeasible, four Maximal Satisfiable Subsystems (MSSes) are found. Each MSS is satisfiable in itself, but including any additional constraints from the system renders it unsatisfiable.

### MUS Enumeration

Since a system can harbor multiple MUSes and MCSes, enumerating these subsets reveals a deeper understanding of the different conflict sources and potential correction plans. Enumerating MUS/MCS is crucial for the comprehensive analysis in program fault detection , circuit design , and explaining machine learning models .

However, MUS enumeration is inherently challenging due to its computational demands. The most naive approach to finding all MUSes is to exhaust the power set of the constraint system, which is impractical for large systems. Advanced algorithms like MARCO  and MUST  utilize heuristics to avoid traversing large blocks of subsets, enhancing efficiency and making the process viable even for complex systems.

This analysis forms a foundational aspect of our discussions on our type error debugging system in Chapter [4](#chap:goanna), providing a structured approach to tackling programming errors through the lens of constraint satisfiability.

## Categories of Type Error

Building upon the introduction of the Minimal Unsatisfiable Subset (MUS) discussed earlier, we revisit and refine the 3 categories of type errors proposed in Chapter [1](#chap:introduction). This categorization directly relates to the constraint representations of type errors and the number of MUSes found within the system. This connection aims to bridge the gap between the intuitive debugging approaches of human programmers and the formal analysis offered by constraint satisfiability.

### Multi-step Type Error

|  |
| --- |
|  |

Illustration of 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).

Multi-step type errors involve a sequence of logical deductions that link one conflicting location to another. When viewed as constraint system, a multi-step type error is a type error 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 Fig. [2.11](#fig:multi-step-2), the MUS consists of the constraints . Removing any one of the elements would allow the program to type-check successfully.

### Multi-witness Type Error

|  |
| --- |
|  |

Analysis of a multi-witness type error, where the case expression could be interpreted as type Char or String. This scenario showcases a type conflict 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.

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 witnesses on one side and on the other, there will be a total of MUSes. For example, in Fig. [2.12](#fig:multi-witness-2), there are three MUSes: . It is crucial to note that the constraint C3 appears in every MUS, signaling a likely root cause.

### Multi-party Type Error

|  |
| --- |
|  |

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.

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 example shown in Fig. [2.13](#fig:multi-party-2), there are three MUSes: . Unlike the multi-witness scenario, no single element appears across all MUSes, suggesting a more complex type conflict scenario.

These categories of type errors illustrate the main challenges in explaining type errors effectively:

1. **Multi-step Type Errors:** It is beneficial to clarify the interdependent relationships between each pair of constraints sharing the same logical variable.
2. **Multi-witness Type Errors:** Identifying the number of witnesses and their associated source code locations on each side of the conflict can be insightful.
3. **Multi-party Type Errors:** Breaking down these errors into simpler forms can aid in understanding and resolving them.

It is important to recognize that these categories are not mutually exclusive; a type error may exhibit characteristics from multiple categories, necessitating a combination of approaches to fully elucidate the underlying issues.

## Conclusion

This chapter began by establishing a historical context for Haskell, highlighting its significant role in both educational settings and programming language research. We explored various methods of type checking and inference within Haskell, starting with an introduction to the Hindley-Milner type inference system and Algorithm W. Although Algorithm W is a well-established and formally proven method, it is not without its shortcomings, particularly its biased nature in handling type errors. We discussed how Algorithm M addresses these limitations by adopting a top-down approach to type checking.

Further, we delved into type error slicing, a technique that identifies all relevant locations of a type error through the use of minimal unsatisfiable subsets. We also introduced interactive type debugging, which builds on the foundations of type error slicing, offering more advanced post-hoc analysis using MUS and allowing programmers to interactively query and explore type errors, thus providing a more engaging and dynamic debugging experience.

We highlighted the fundamental tools and frameworks that underpin type error slicing and interactive debugging, setting the stage for the exploration of my contribution (ChameleonIDE and Goanna) following this line of research in subsequent chapters. Finally, we revisited and refined our categorization of type errors, now defined in terms of minimal unsatisfiable subsets, providing a clearer and more structured framework for understanding and addressing these errors.

# ChameleonIDE: Interactive Type Error Exploration

This chapter explores the challenges associated with traditional tools for debugging type errors in Haskell. We focus on their limitations in identifying all relevant locations, explaining errors in user-friendly language, and the tendency to introduce bias. We introduce ChameleonIDE, a type error debugging tool aimed at exploring multi-step type errors efficiently and intuitively. ChameleonIDE addresses the common challenges of traditional compiler tools and presents type errors in an optimized, comprehensible manner. We delve into the design of ChameleonIDE and demonstrate its usage through examples of resolving type errors. Finally, we present our user studies designed to evaluate the effectiveness of type error visualization and interactive debugging tools.

This chapter uses the content from our paper *ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration* , with slight modifications to ensure coherence within this thesis. This work was published and presented at the International Conference on Program Comprehension (ICPC) 2023.

## Introduction

Functional programming languages such as Haskell, and ML have long enjoyed rigorous type systems and expressive type-level features. Techniques such as type inference and algebraic types have been standard practice for decades in these languages, and more recently in multi-paradigm languages, such as Rust and TypeScript. Various type system advances were introduced in Haskell and ended up in mainstream languages years or even decades after, leading many to consider Haskell the “type-system laboratory" . Type classes, an implementation of generic programming, were introduced to Haskell in 1988 , and now can be found in most popular languages such as C# , Java , and TypeScript .

One crucial challenge of programming in Haskell, and many other statically-typed languages, is that type errors can sometimes be difficult to resolve . In particular, they may point to locations that are not the root causes of the type error, expose errors in cryptic language, or provide misleading fixing suggestions .

We introduces ChameleonIDE, an interactive type debugging tool for Haskell. It can visualize the relevant context of a type error: where it happens or could have happened and which parts of the code cause it. In addition, ChameleonIDE allows programmers to interactively explore all the parts of code where multiple types can be inferred and to resolve ambiguity. The most noticeable features are the type compare tool (Section [3.3.1.0.1](#sub:type-compare)), the candidate expression card (Section [3.3.1.0.2](#sub:candidate-expression)), and the deduction step (Section [3.3.1.0.3](#sub:deduction-steps)). These features are integrated into a debugging environment and can be enabled or disabled separately based on the programmers’ preferences and debugging needs. ChameleonIDE is open-source and is available at  .

This paper makes the following contributions:

* We provide the design and implementation of the ChameleonIDE to visualize the relevant context of a type error and allow programmers to explore and verify the error locations in small chunks interactively.
* We report the results of three experiments designed to evaluate ChameleonIDE.

Our experiments showed that programmers using ChameleonIDE fix type errors faster than with traditional text-based error messages. This difference is more significant when solving harder tasks. Further, programmers who actively use ChameleonIDE interactive features fix type errors faster than simply reading the type error output. Although ChameleonIDE is designed to work with the Haskell language, we plan to extend the underlying ideas to work with other strongly typed languages, such as Rust or TypeScript.

## Motivation

The design requirements of ChameleonIDE are motivated by limitations of traditional type errors, as documented in a number of studies (e.g. ), but which we illustrate here with a few motivating examples.

![](data:application/pdf;base64,)

A type error is displayed in Visual Studio Code with the Haskell Language Server support. The expression addPair is blamed for causing the type error. This may not match the programmers’ intention.

**Traditional type errors show only limited location** Haack and Wells  noted that “*Identifying only one node or subtree of the program as the error location makes it difficult for programmers to understand type errors. To choose the correct place to fix a type error, the programmer must find all the other program points that participate in the error.*” The type error in Fig. [3.1](#fig:motivation-example) can be fixed in multiple locations. For instance replacing [’0’..’9’] on line 1 with [0..9], or replacing fst x and snd x on line 2 with read (fst x) and read (snd x). In the type error message, only the addPair expression on line 4 was blamed. The whole context is visible in this small example, but it can become problematic in large programs where the lines contributing to the type error are far apart in the source code.

**Traditional type errors are biased** A common form of bias happens when a type error is reported in one expression, but it can occur in multiple other expressions as well. In Fig. [3.1](#fig:motivation-example), the error message arbitrarily focuses on only addPair, while ignoring that the literals in the definition of u may be incorrect. Another form of bias is that traditional type errors are often framed as conflicts between Expected type and Actual type. This framing is standard practice in most typed languages. However, what is expected and what is actual are a side effect of different unification orders rather than the intention of the programmer. In both forms, the error message may lead programmers to falsely believe the validity of parts of code and wrongly accuse others.

**Traditional type errors give poor explanations**

When the compiler type checks a program, it generates a series of constraints from the source code and searches for a model that satisfies all these constraints. However, the details of this process are hard to explain to users and are usually not reported by compilers. For the typical type error shown in Fig. [3.1](#fig:motivation-example), the evidence for the type error is gathered from the previous declarations. These have to be rediscovered by programmers using less rigorous methods.

### Design Goals of ChameleonIDE

Based on the limitations of traditional type errors, we give the following design requirements for ChameleonIDE:

**Show** all the possible locations where the type error happened or could have happened.

**Explain** type errors, avoiding jargon and internal constructs of the type checker.

**Do not presume** which expression is to blame for the type error based on the order of computation or which possible type for an expression is ‘actual’ or ‘expected’.

## Chameleon IDE

![](data:application/pdf;base64,)

**The anatomy of ChameleonIDE.** The editor pane (left) is similar to a traditional code editor. Fragments of source code may have a highlight color (A). Additionally, an explanation layer (B) displays if deduction steps are enabled. The debugging pane contains three blocks. First, the error statement block contains an error statement (D), optionally, a list of candidate expression cards (E), a list of deduction steps (F), and a control bar (G) to increment/decrement deduction step. Second, the conflicting types block shows two alternative types (H). Third, the relevant type information block shows additional information (I) that may help understand type errors.

ChameleonIDE comprises two parts: a type inference engine and a novel interactive debugging interface. The debugging interface is designed from the ground up; the type inference engine is a re-implementation of the original Chameleon with several novel improvements, as described in Section [3.3.2](#sec:typeinferenceengine).

### The Debugging Interface

The ChameleonIDE debugging interface provides three main features to visualize and explain type errors.

##### Type compare tool

The type compare tool shows conflicting types in different colors, each type associated with one or more error locations highlighted in a matching color (Fig. [3.3](#fig:compare)). If the programmers know the expression’s intended type (they usually do), they will be able to eliminate half of the possible locations. A hover interaction over one of the possible types facilitates such bisection, causing only the relevant locations that contribute to that type to be highlighted.

![](data:application/pdf;base64,)

**ChameleonIDE with type compare tool enabled**. ChameleonIDE identified the conflicting types for the expression u and associated the relevant locations with each type. Compare the output with the traditional type error message in Fig. [3.1](#fig:motivation-example).

![](data:application/pdf;base64,)

**ChameleonIDE with candidate expression cards enabled.** Indicates the type error can occur in the definition of x or y.

![](data:application/pdf;base64,)

**ChameleonIDE with deduction steps enabled.** ChameleonIDE explains the type error in four steps. In the screenshot, the active step is step 2, where ChameleonIDE shows that the expression x and y should have the same type.

##### Candidate Expression Cards

A candidate expression is an expression that can be inferred to have two conflicting types. When a type error is detected, ChameleonIDE provides a list of all candidate expressions, and programmers are free to choose the problem to resolve by clicking on one candidate expression card. In the example shown in Fig. [3.4](#fig:expression), x and y are both candidate expressions. Fixing either type error can make both expressions well-typed.

Programmers select a candidate expression card by clicking on one card. Once a card is selected, the information in the conflicting types block changes to reflect the change of candidate expression. In the editor pane, some error locations change highlight colors based on the updated candidate expression. Alternatively, programmers can preview the change of a candidate expression by hovering on one card. The hover effect is reverted once the cursor moves away.

##### Deduction steps

Deduction steps allow programmers to explore all the error locations one at a time (Fig. [3.5](#fig:deduction)). Steps are shown as a list of sequentially numbered circular buttons (step buttons) and an explanation layer in the editor window. In the explanation layer, the two locations under examination are outlined, and a line is drawn to connect these two locations. This line is accompanied by a human-readable text explanation of their semantic connection. Programmers are free to activate any step. The active step is shown in green. When activating a step, some highlights switch color. The message in the explanation layer changes accordingly. A program in Fig. [3.5](#fig:deduction) generates a list of steps shown in Fig. [3.6](#fig:step-interface) left.

Programmers can use mouse and keyboard shortcuts to increment or decrement the step number or jump to any step. Programmers resolve type errors by navigating through all the deduction steps and verifying whether each explanation aligns with their intention. Eventually, they will find a step that does not match, and the type error can be fixed by modifying one of the two outlined locations.

![](data:application/pdf;base64,)

Deduction steps if they are shown all at once. In practice, steps are shown one at a time. Programmers increment or decrement the step number using the step control bar (Fig. [3.2](#fig:anatomy)-G) or by directly clicking on a step button (Fig. [3.2](#fig:anatomy)-F). To increment or decrement the deduction step can be intuitively thought of as moving the position of the *splitting point* (dotted lines) where the blue and orange highlights divide.

Internally, deduction steps are different ways to divide the error locations into two groups, denoted by the two colors. Each color infers a different type of candidate expression. Each increment/decrement of the step changes the splitting point (dotted lines in Fig. [3.6](#fig:step-interface)) of the two colors. Deduction steps are an interactive view of multi-step type errors.

##### Multiple Modes

Nielson pointed out that the two most important issues in designing for usability are understanding the users’ tasks and the differences in users . From analyzing how users use ChameleonIDE, we realized that the ideal debugging interface should adapt to the specific programmer and programming task. There are cases where a programmer wants the debugger to simply “show the answer", and others to dive deeper into the problem domain and search for the optimal solution. To accommodate the need to customize the level of information density and granularity of control, ChameleonIDE provides three modes: basic, balanced, and advanced. Programmers can switch between modes by clicking on the mode switching toggles (Fig. [3.2](#fig:anatomy)-C). The features accessible from different modes are summarized in Table [3.1](#tab:chameleon-features).

ChameleonIDE modes and features

| *Mode* | *Features* |
| --- | --- |
| Basic Mode | Type Compare Tool |
| Balanced mode | Type Compare Tool |
|  | Candidate Expression Cards |
| Advanced mode | Type Compare Tool |
|  | Candidate Expression Cards |
|  | Deduction Steps |

### The Type Inference Engine

Chameleon was originally a command-line tool developed in the early 2000s to improve type error reporting for the Haskell programming language. Unlike traditional type errors produced by the Glasgow Haskell Compiler (GHC) , which uses a Hindley–Milner type inference system, Chameleon infers types using constraint solving. In Chameleon, constraints are generated from the source code based on typing rules. In addition, each constraint is labeled with the location where it is generated. This set of constraints is consistent if the program is well-typed and inconsistent otherwise. When a type error occurs, an efficient algorithm is used to derive a minimal subset of the constraints that still contain inconsistencies. This subset is called a Minimal Unsatisfiable Subset (MUS). From this, Chameleon can report a list of locations, using the labels of constraints that are in the MUS. Stuckey, Sulzmann and Wazny showed that program locations linked to the constraints from an MUS are all relevant to the type error and must include the cause of the error .

Despite successfully borrowing the underlying ideas, we could not reuse the original implementation of Chameleon since the project language standard and libraries used were out of date. ChameleonIDE implementation extends the original Chameleon approach in a number of ways. ChameleonIDE supports the base language of Haskell2010 . However, ChameleonIDE does not support any advanced language extensions, such as GADTs.

##### Recovering concrete types from type errors

Using only constraints from the MUS is sufficient to locate the type error, but to recover types from type errors, we need constraints from parts of the program that are irrelevant to the type error. For instance, consider an ill-typed 2-tuple where two possible types can be assigned: (Int, Int) and (Int, String). The types reconstructed from Chameleon may be (a, Int) and (a, String). Although the recovered types are theoretically correct, they introduce the notation a, which denotes a generic type variable that can be any type, making the error message harder to understand. To solve this issue in ChameleonIDE, for each constraint c in the MUS, we find a maximally satisfiable subset (MSS) from all the constraints that contain every other element of MUS but not c. These maximally satisfiable subsets, while not helpful in error localization, will produce the most concrete types, see Fig. [3.7](#fig:compare-to-original). Concrete types, such as Int and String, often provide extra information to programmers. With a type of (Float, Float), programmers may want to convey a point in 2D space. However, a type of (a, Float) does not preserve such information.

![](data:application/pdf;base64,)

Reporting the same type error, Chameleon uses more abstract types Int -> a and Char -> a, while ChameleonIDE uses the concrete types (types that do not contain type variables) Int -> Bool and Char -> Bool.

##### Type error explanation

In addition, ChameleonIDE provides support for type explanation. Similar to the type explanation system in  , ChameleonIDE is able to produce a human-readable explanation, but for type errors. This is achieved by annotating nodes in the abstract syntax tree with constraints and the type inference rules used. We generate an inference history from constraints and accompanying annotations.

![](data:application/pdf;base64,)

A simple program that is ill-typed. It generates two constraints from line 1 and one constraint from line 2.

For instance, for the program in Fig. [3.8](#fig:listing1), ChameleonIDE generates the following constraints and labels (in brackets) (if condition), (if branches), (definition). Clearly, as can not unify with both *Bool* and *String*, this program is not well typed. ChameleonIDE can construct a human-readable explanation from the MUS. An example output for Fig. [3.8](#fig:listing1) can be: a has type Bool because a is the condition of an if statement; however, a has type String because a is defined as the string literal "True". This explanation facilitates the deduction steps (Section [3.3.1.0.3](#sub:deduction-steps)).

## Walkthrough

In this section, we showcase ChameleonIDE by walking through examples of its use. The examples are given from the perspective of a hypothetical Haskell programmer Maxine.

### Basic mode

Maxine writes a function to calculate the sum of a list of numbers, but ChameleonIDE shows there is a type error (Fig. [3.9](#fig:basic-mode-1)). After reading the error reports, Maxine realizes that the error revolves around the expression xs. That is: xs can be either [a] or Int. By matching the color in the conflicting type block (Fig. [3.2](#fig:anatomy)-H) and the highlighted error locations Maxine knows that the [a] results from the pattern matching of the : operator, while Int results from using + to add two expressions.

![](data:application/pdf;base64,)

Maxine’s code to calculate the sum of a list of integers; ChameleonIDE reports an error on the expression xs.

At this point, Maxine knows that type 1 might align with her intention, and therefore, the error locations with blue highlights must be erroneous. After examining the program, it becomes clear that Maxine forgets to apply the sum function recursively at the right-hand side of the addition.

### Balanced mode

Maxine writes additional code to add only even numbers in a list of integers, reusing the sum function she wrote earlier. After saving the file, ChameleonIDE shows a type error in the expression sum (Fig. [3.10](#fig:balance-mode-1)). However, this is not helpful because Maxine has just verified the implementation of sum. Switching to balanced mode, ChameleonIDE shows two cards: sum and evens.

![](data:application/pdf;base64,)

Maxine’s code to calculate only the sum of even numbers. ChameleonIDE reports an error with two candidate expressions.

![](data:application/pdf;base64,)

Clicking on the evens card (5) results in the changes in the conflicting types panel to show the possible types for evens, and the changes highlight color to reflect the assumption that the definition of evens is the cause of the error.

Maxine therefore clicks on the evens card and ChameleonIDE reports two possible types for the expression [Int] and [Int] -> [Int] (Fig. [3.11](#fig:balance-mode-2)). Knowing the expression evens holds a temporary list of even integers (hence it is of [Int] types), Maxine concludes that the Possible type 2 is unintended. The locations with blue highlights must contain the cause. It does not take long for Maxine to realize the list l is not supplied to the filter function.

### Advanced mode

![](data:application/pdf;base64,)

Maxine’s code to calculate only the sum of even numbers in advanced mode. The current step is step 5, ChameleonIDE explains that the two appearances of expression evens should have the same type.

![](data:application/pdf;base64,)

In step 6, ChameleonIDE explains that evens is defined as the expression filter isEven. The left-hand side and the right-hand side should have the same type.

![](data:application/pdf;base64,)

In step 7, ChameleonIDE explains that filter is applied to the function isEven. Assisted by the type of filter in the Relevant Type Information panel on the bottom right, Maxine can find the type error that filter expects two arguments but receives one.

To illustrate the deduction steps with the task shown in Section [3.4.2](#sub:balanced), first, Maxine clicks on step 5 (Fig. [3.12](#fig:advanced-mode-step5)) and verifies that the two occurrences of evens are supposed to be identical, and the second use means evens is a list of integers. Second, she clicks on step 6 (Fig. [3.13](#fig:advanced-mode-step6)) and verifies that evens should be the same type as the declaration on the right-hand side.

Lastly, Maxine clicks on step 7 (Fig. [3.14](#fig:advanced-mode-step7)), and it shows that the filter function is applied to one argument isEven. By consulting the relevant type information, Maxine identifies that filter is expecting two arguments while only one is provided.

## Evaluation

We conducted three user studies, iteratively refining the ChameleonIDE UI and evaluating several research questions as per Fig. [3.15](#fig:timeline).

![](data:application/pdf;base64,)

The timeline of ChameleonIDE evaluation.

### Experiment Design

#### **Recruitment**

Participants were recruited via the Reddit *r/haskell* and *r/programminglanguages* communities. Participation is fully anonymized; detailed ethical implications of these experiments are reviewed and approved by the IRB of the authors’ institution.

#### **Experiment setting**

Experiments were conducted online and unsupervised. All user studies use a web-based debugging environment developed by the authors.

#### **Training and group assignment**

After consent, participants received interactive training on the tool interface and interactive features. Participants were also shown a cheat sheet summarizing the key functionality of the interface and had access to the cheat sheet at all times during the study. Participants were given 4 trial runs (2 for each setting) before the data collection started. All the studies used a within-subject design to evaluate the effectiveness of different tools or feature sets while counterbalancing the differences in programming proficiency between participants. In each study, participants were required to complete a series of programming tasks (8 for studies 1a and 1b, 9 for study 2). At each task, a participant received a single Haskell file containing one or more type errors. They were then asked to correct the code using the given tool.

#### **Data Collection**

Time is measured from the start of each task to the first time the program is successfully type-checked and passes all the functional tests. Participants can skip a task if they are stuck. After completing all tasks, participants are prompted to complete a debriefing survey. The survey questions include their Haskell experience and feedback on the tools.

We used a browser session recording tool  to record the study sessions. This allows us to identify usability issues in the study and to recognize general patterns.

### ChameleonIDE Human Studies

#### ChameleonIDE 1

An earlier version of the UI than that depicted in Figs. (2-13), it featured the type inference engine that recovers most concrete types after type errors occur and a minimal set of debugging features. Key features in ChameleonIDE 1 include showing two (or more) alternative types, showing all possible error locations, dividing possible error locations into groups based on alternative types, and concrete type restoration. In short, ChameleonIDE 1 is equivalent to ChameleonIDE 2 set to basic mode.

Two studies (1a & 1b) were conducted to compare the effectiveness of solving type errors using ChameleonIDE 1 and GHC compiler error messages. We chose GHC compiler error messages as the baseline because it is the canonical tool for working with type errors in Haskell.

Eight tasks were given in both studies. In study 1a, the tasks were taken from the exercises of the Haskell programming class in the authors’ institute. In the second study, the tasks are sourced from the top 20 Haskell topics on GitHub . The authors then manually added type errors into the program. In both studies, the type errors include simple mismatch, confusing syntax, missing instance, precedence and fixation, infinite types, and confusing list versus element. These categories follow the common type errors in Tirronen’s study .

Studies (1a & 1b) address the research question:

**RQ1.** *Do programmers solve type errors faster with ChameleonIDE than GHC compiler error messages?*

![](data:application/pdf;base64,)

Study 1a task completion time (secs.) with 95% confidence interval.

![](data:application/pdf;base64,)

Study 1b task completion time (secs.) with 95% confidence interval.

#### **Results**

The data collected during study 1a, Fig. [3.16](#fig:analysis-1a) does not show significant differences across Tasks 1-7. In hindsight, these tasks were trivial challenges for most users, and the individual differences among participants are generally more significant than the differences between treatments. However, one interesting observation is task 8, where the ChameleonIDE group outperformed the GHC group. We attribute this significant difference to the difficulty of Task 8. The source file is longer and involves more language features (abstract data types and high-level functions). GHC struggles to produce a relevant error message for this type of error. From this result, we hypothesized that we might observe a more significant difference using tasks with lengthier and more realistic source code. This hypothesis is also supported by the most common feedback claiming that the tasks were too trivial to invite meaningful evaluation. One participant said, “Looks nicer than GHC, but without trying it on something more complicated, I cannot conclude whether it would help me in practice."

Therefore, in study 1b, we introduced more difficult challenges and indeed observed that the ChameleonIDE group was faster than the GHC group in almost all tasks (Fig. [3.17](#fig:analysis-1b)), barring task 1. A two-sample paired t-test was performed to compare the completion time between ChameleonIDE and GHC groups. There was a significant difference between the two groups: . For task 1, it is suspected that some participants spent more time exploring the interface of ChameleonIDE due to its unfamiliarity. For all other tasks, from the video recordings, we saw many ChameleonIDE users confidently skip reading unrelated chunks of code, while GHC users generally read through the whole program. In harder problems and messier code, we notice programmers start to report the benefits of ChameleonIDE. *“It’s most useful feature that I noticed was that it points out the locations of both conflicting uses; GHC often makes it difficult to figure out how it’s coming to a conclusion about a type."* reported one participant. *“I think ChameleonIDE does a much better job than GHC’s error messages. I like that it shows the sources for the type judgments. This makes it quite easy to figure out how to rectify errors."* reported another participant.

#### ChameleonIDE 2

Based on observations of Study 1, we introduced several new features to ChameleonIDE, eventually resulting in the UI depicted in Figs. (2-13). Interactive features were available in this iteration, such as deduction steps, candidate expressions, and mode switching. A few other user interfaces were designed and prototyped between the development of ChameleonIDE 1 and ChameleonIDE 2. Study 2 addresses the research question:

**RQ2:** *How do programmers use the interactive features in ChameleonIDE 2?*.

More specifically:

* **RQ2.1** How do programmers use the advanced features provided by ChameleonIDE 2?
* **RQ2.2** Do programmers prefer switching modes during debugging type errors?
* **RQ2.3** What are programmers’ preferences among the three modes provided by ChameleonIDE 2?

During each run, the initial mode of each task alternated between the three different modes, repeating three cycles in nine tasks. The order of the three modes in each cycle was counterbalanced among all participants. However, participants could switch to other modes at any time.

#### **Results**

Study 2 is more exploratory in methodology than Study 1. We encouraged programmers to discover their way of using the tool. In post hoc analysis of the collected log data, we were able to extrapolate some interesting patterns of how the tool was used.

**RQ2.1**. The most striking feature of the data is that users tend to vary wildly in their use of the tool. Some users used the features extensively, while others completed the tasks without actively exploring the given information. Based on this discrepancy, we divided the users into three groups in table [3.2](#tab:interaction-level).

Levels of programmer interaction and their description

| =.26Interaction level | =.74  Description |
| --- | --- |
| =.26*Minimal* | =.74  Users completed the tasks by making changes in source code, type checking, and reading error messages. |
| =.26 *Low* | =.74  Users only actively used universal features in all modes, for example, hovering on "Possible type 1" and "Possible type 2" to narrow down error space. |
| =.26 *High* | =.74  Users did everything from the low interaction group but used features specific to the Balanced mode and the Advanced mode, such as activating steps and expression cards. |

As shown in Fig. [3.18](#fig:r4-analysis), the time to complete each task roughly relates to the interaction level of participants. Participants with higher interaction levels generally performed better, and the lowest interaction level was worse. Tukey’s HSD Test for multiple comparisons found that the completion time was significantly different between the minimal interaction group and the high interaction group (, 95% C.I. = [18.26, 31.41]), and between the minimal interaction group and the low interaction group (, 95% C.I. = [11.96, 26.67]). The results from three tasks stand out from the general trend: in Tasks 4 and 6, higher interaction users performed worse, and in Task 9, the general trend is exaggerated. As with Study 1a and 1b, this difference is likely related to task difficulty. Tasks 4 and 6 are shorter than other tasks. The ideal fixes for these two tasks are placed relatively early in the source code (both in the first two lines of the source code). Users simply reading top to bottom could quickly identify the error without needing to skip unrelated sections of code using the information provided by ChameleonIDE. This reduced the apparent benefit of ChameleonIDE in these tasks. On the other hand, task 9 is the lengthiest task of all. It also involves deeply nested type definitions that are harder to follow in mind.

![](data:application/pdf;base64,)

Study 2 task completion time (secs.) with 95% confidence intervals.

Another observation is when using the mode switching feature of ChameleonIDE, we show this by presenting the starting mode and finishing mode of each task and each participant in a correlation matrix (Fig. [3.19](#fig:r4-mode-switching)). This observation suggests two characteristics of using multi-mode debugging tools. First, to answer **RQ2.2**, programmers are roughly split in this matter: 53% changing modes vs. 47% staying in the same mode. Second, to answer **RQ2.3** when changing modes, programmers generally switch to the more informative modes instead of the more concise ones.

![](data:application/pdf;base64,)

Study 2 mode switches by starting mode. Users overwhelmingly switched to the more sophisticated interface mode.

### Limitations

One threat to the validity of the evaluation is the number of participants. Although we received hundreds of online participants for each study, the studies suffered from a high abandonment rate (especially study 1b). This was expected: the programming challenges are difficult, and our volunteer participants are unremunerated.

Because we recruited participants online and anonymized all the participants, it is possible for participants of a previous study to enter a later one. This creates variation in familiarity. We offset this by using new code challenges in every study and conducting trial runs before data collection to bring new participants up to speed. Conducting studies remotely and unsupervised left us no means to intervene when users encounter usability issues. To mitigate this, we conducted cognitive walkthroughs and sandbox pilots before running each study.

Future evaluation would benefit from using more realistic tasks. The tasks in our human studies do not get as complex as professional Haskell programmers may face in a typical production codebase. It would be interesting to see how ChameleonIDE is used against type errors that span multiple files and packages and include more confusing abstractions, like Monads, Monad transformers, and Lenses.

## Discussion

This chapter presents the interactive type debugging tool ChameleonIDE and charts the evolution of its design across several iterations in response to user evaluation and feedback, as well as examines the effectiveness of the general approach compared to traditional static type error messages. We found that programmers using ChameleonIDE are able to debug errors faster than using traditional text-based error messages. This effect is shown more clearly when the task is not trivial. We found that programmers who actively use ChameleonIDE’s interactive features are more efficient in fixing type errors than passively reading the type error output. In this section, we discuss a few interpretations of the results.

### Effect on Reading Source Code

From the results of Study 1a, we observed that the choice of debugging tool had little effect on how fast programmers solve simple type errors. Conversely, when facing more realistic problems (longer source code, error locations more scattered) in study 1b, programmers are more effective using ChameleonIDE. One explanation is that ChameleonIDE reduces the amount of reading time by taking programmers more directly to the problem. Earlier studies showed that reading source code is generally the initial step of solving programming problems and is done in several passes. Although traditional compiler error message tools initially show fewer locations, these may be incomplete, meaning that programmers have to expand the reading span without clear guidance. In contrast, ChameleonIDE shows more error locations initially. However, the completeness of error locations assures programmers which part of the source code can be safely skipped.

### Forming Debugging Plans

From the results of Study 2, we found that programmers who use the interactive tool fix type errors faster than the ones who passively read the error output. This effect is stronger in harder tasks. We speculate that one factor of this result is that ChameleonIDE helps to develop debugging plans. We observed that when working with ChameleonIDE, programmers form different debugging plans to attack the problem. Among the *high* interactivity participants in user study 2, some programmers cycle through deduction steps as a guide to reading source code; some navigate to both ends of the deduction chain where types are normally grounded and concrete. In contrast, *minimal* interactive participants generally form similar plans, including carefully reading the program text and manually annotating expressions based on their understanding of the program.

### Externalize Intermediate Typing Information

We speculate another factor of the effectiveness of ChameleonIDE interactive debugging tools is that they help programmers effectively chunk intermediate information. With the program shown in Fig. [3.20](#fig:listing2), ChameleonIDE offers two candidate expressions: f can be typed as Int -> Bool or Char -> Bool; z can be typed as Int or Char. Although these two statements are equivalent in theory, programmers are often required to compute the latter from the former or vice versa. This computation may carry out multiple layers. Programmers have to remember all the intermediate types and their reasoning throughout such mental gymnastics. Assisted by candidate expression cards and deduction steps, this intermediate information is externalized on screen and can be retrieved anytime. A recent study on working memory suggested this approach may provide a positive effect in helping programmers manage cognitive load and free up working memory space for high-level thinking.

### Exploreg Multi-step Type Errors

ChameleonIDE’s type debugging features are intentionally designed to facilitate the understanding of multi-step type errors. During a type error, ChameleonIDE finds a single MUS and uses each constraint inside the MUS to generate interactive debugging steps, allowing programmers to “replay” how type errors occurred in a “frame by frame” fashion. This design faithfully presents the internal logic conflict to programmers. It aligns with our analysis and debugging strategy outlined in Chapter [1](#chap:introduction) by visualizing the chain of logical inference and allowing programmers to interact with it freely. However, it is worth noting that ChameleonIDE does not offer additional insights into multi-witness and multi-party type errors due to its focus on a single MUS.

![](data:application/pdf;base64,)

ChameleonIDE reports an error in the expressions f and z

## Related Work

In this section, we position ChameleonIDE in three fields of study that are highly relevant to its development: program slicing, compiler error message enhancements, and interactive debugging.

### Finding all type error locations

Many have studied the approach of finding all locations that contribute to a type error . Type error slicing  is a technique that finds locations that are complete and minimal for the type error. Internally labeled constraints and Minimal Unsatisfiable Subset (MUS) generation are used to generate these slices. The language supported in Haack’s work was a subset of Standard ML. The original Chameleon  used Constraint handling rules (CHR) to support the computing of type error slices in Haskell. Chameleon also supported advanced type-level features (type classes and functionally dependent types). The project also introduced the ability to query type information through a command line interface. Although Chameleon was firmly grounded in results from type theory, its designs were never evaluated with user studies. While finding all error locations is useful in comprehending type errors, it is only 1 of the 7 properties listed in the proposed manifesto of good type error reporting . To the best of our knowledge, ours is the first user-centered evaluation of an interactive type debugging system involving type-error slicing.

### Producing high-quality error explanation

One weakness of compiler error messages, in general, is that they fail to explain the error in human language. A study on how programmers read compiler error messages  concluded, “Error messages appear to take the form of natural language, yet are as difficult to read as source code." A well-studied approach to producing better error explanations is through ECEM (Enhanced compiler error message). Through a series of mixed-method studies, Prather showed  that ECEM has a positive result in understanding compiler errors. Decaf  is a tool that can rephrase Java compiler error messages into an enhanced version. In a study of over 200 CS1 students, Decaf was shown to reduce overall errors in their coding practices. Berik proposed a framework  for constructing compiler error messages based on argumentation theory and showed that error messages following a simple argumentation layout or an extended argumentation layout are more human-friendly. These works show the significance of improving the language in the compiler error messages. Most principles and suggestions are followed in ChameleonIDE in constructing error statements. However, these earlier studies targeted not type errors alone but general compiler errors (some even included runtime errors). The nuances of type errors, such as alternative typing, were not considered. Moreover, these explanation systems were designed specifically for novice users.

### Interactive Debugging

Modern programming tools can offer alternative methods of code authoring, display real-time feedback, and reveal complex programming contexts through visualizations. Many tools aim to improve the debugging experience using such capabilities. We list two. Hazel Tutor is an interactive type-driven environment for the OCaml language. It can automatically fill type holes by suggesting template expressions (called “strategies” by the authors) through a popup window. It also provides a cursor-based type inspector that allows programmers to query the types of different parts of the program. Whyline  is a Java debugging system that allows a user to ask questions like “why does variable X have value Y." It also allows users to interactively ask follow-up questions to gain further knowledge of the nature of an error. These debugging tools are important motivations for developing ChameleonIDE. However, they focus on different aspects of the debugging process. Java Whyline mainly tackles the problem of unintended runtime behavior, while Hazel Tutor specializes in development assistance supported by type holes.

## Conclusion

In this chapter, we present ChameleonIDE, a type debugging tool for the Haskell programming language. Its constraint-based type inference engine provides unbiased and comprehensive error location reporting. Our studies evaluated the tool’s design with programmers. We found that, particularly for more complex tasks, ChameleonIDE helped programmers to fix type errors more quickly than traditional text-based error messages. Further, programmers actively using ChameleonIDE interactive features are shown to fix type errors faster than simply reading the type error output. ChameleonIDE currently works with the Haskell language, but in the future, we plan to extend the type-checking system to work with other strongly typed languages, such as Rust or TypeScript.

# Goanna: Finding All Type Errors Using Minimal Correction Sets

In this chapter, we introduce Goanna, another Haskell type error debugging tool. Goanna aims to provide a set of debugging utilities to support resolving multi-witness type errors and multi-party type errors. Goanna sets itself apart from both traditional tools and ChameleonIDE through its unique features, such as identifying all potential causes of type errors, suggesting fixes, and supporting cross-module debugging. This chapter begins by highlighting the limitations of conventional compiler error messages and then details Goanna’s capabilities in identifying and ranking potential causes by their likelihood. We then discuss the implementation strategies and heuristic methods employed in Goanna. Finally, we present an empirical evaluation of the system’s accuracy, conciseness, and performance using real-world Haskell code examples.

This chapter uses the content from our paper *Goanna: Resolving Haskell Type Errors With Minimal Correction Subsets* , with slight modifications to ensure coherence within this thesis. This work was publicly available as a preprint when submitting this thesis.

## Introduction

Statically typed languages have gained popularity in the mainstream programming world . Many new languages have been designed with strict type systems, while others have introduced static typing through external tools. Numerous studies indicate that programming with statically typed languages can prevent certain errors , enhance code quality , and reduce maintenance costs compared to similarly positioned dynamic languages . Despite their increasing popularity and benefits, challenges persist in the real-world adoption of these languages . The steep learning curve of complex type systems remains an obstacle to their adoption.

Haskell is renowned for its expressive and robust type system. It enables programmers to model complex problems as constructs and relations within type systems and develop programs in a type-driven style. Historically, many type system innovations initially introduced by Haskell , including algebraic data types, type inference, and type classes, have now found their way into mainstream programming languages .

However, Haskell is also known for its steep learning curve and unforgiving type errors. Numerous research efforts have attempted to address these challenges . The type errors generated by the most commonly used Haskell compiler, GHC (Glasgow Haskell Compiler), often lead to confusion among novice users, and sometimes experts. For instance, in the program shown in Fig. [4.1](#fig:motivation), a type mismatch between a Char type and an integer number type results in a perplexing type error for novice users. We have identified three challenges for making use of these error messages:

1. Fixated on one possible cause while other potential causes exist.
2. Changing the suggested location does not completely rectify the error.
3. Not enough contextual information for programmers to understand how the judgment was made.

|  |
| --- |
|  |

Inspecting a type error using the Haskell compiler GHC (Glasgow Haskell Compiler) version 9.0.2

To address these challenges of diagnosing and fixing type errors in Haskell, we present a new tool: *Goanna*. Goanna is a Haskell type checker based on Minimal Correct Subset (MCS) enumeration. Compared to traditional type-checking tools, Goanna provides improved type error reporting by giving a comprehensive list of possible causes and suggesting valid fixes for each cause. Goanna differs from past type debugging systems (as reviewed in Section [4.6](#sec:related-work)) through its use of Minimal Correction Subsets (MCS), where a single MCS represents a complete set of locations that constitutes a possible cause.

To further enhance Goanna’s support for type-error resolution, we provide optimization strategies (Section [4.3.4.3](#sub:optimization)) to identify and reduce the unhelpful suggestions, as well as ranking heuristics (Section [4.3.4.2](#sub:ranking)) to suggest more likely fixes first. Additionally, we provide Goanna-IDE, an interactive debugging front-end designed to efficiently navigate and interpret Goanna’s type error diagnosis.

We conducted empirical studies that evaluated Goanna’s accuracy (Section [4.4.2](#sub:eval-accuracy)), conciseness (Section [4.4.3](#sub:eval-conciseness)), and performance (Section [4.4.4](#sub:eval-performacne)). Our evaluation shows that compared to other type-checking tools, Goanna consistently provides accurate error diagnostics and correct fixes in its top suggestions. We also demonstrate that Goanna generally offers a concise list of possible causes, thanks to its cause optimization process. Although Goanna may not consistently provide instantaneous results for real-time feedback, it can deliver on-demand diagnoses when programmers require additional assistance.

The key contributions of this research include:

* Goanna, a Haskell type checker with improved error reporting based on MCS enumeration and program slicing;
* Goanna-IDE, an interactive type error debugging interface for Haskell;
* A collection of heuristics and optimization techniques to enhance MCS-based type error reporting; and
* An evaluation of Goanna’s accuracy, conciseness, and performance.

The techniques we used in Goanna, such as MCS enumeration and heuristics for ranking possible causes, are not exclusive to Haskell but apply to statically typed programming languages in general. We intentionally designed Goanna with a modular architecture that can be easily extended to support other programming languages with similar typing disciplines.

## Goanna-IDE Walkthrough

We first illustrate Goanna’s capability by demonstrating the usage of Goanna-IDE. Goanna-IDE is a type error debugging interface for Haskell. It is designed to efficiently navigate and make use of Goanna’s type error diagnosis through visualization and interactivity. Goanna-IDE provides comprehensive diagnostic error messages for type errors in Haskell and allows programmers to interactively explore their options. An online demo of Goanna-IDE is available for evaluation at . Goanna-IDE includes a file explorer, a text editor, and a debugging panel. Goanna-IDE provides the following 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.
* Allow users to trace type errors across multiple files.

### 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. [4.2](#fig:goanna-example-1)). When focussing on the cause suggested by GHC, instead of highlighting only the literal 1, Goanna reports all 3 literals that needed to be changed all at once if the programmer chooses to address this cause. 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.

|  |
| --- |
|  |

**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. The most likely fix, based on Goanna’s cause heuristics (Section [4.3.4.2](#sub:ranking)), is the Char literal ’1’ on line 5, indicated by the 3 stars (Fig. [4.2](#fig:goanna-example-1)).

By clicking on the most likely cause, Goanna shows different highlights in the editor (e.g., see Fig. [4.3](#fig:goanna-example-2)). Goanna reports the error is caused by the literal ’1’ and suggests changing 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.

|  |
| --- |
|  |

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

### Identifying all type errors

A key feature of Goanna is its ability to detect all type errors in the code thanks to its MCS enumeration (Subsection [4.3.3](#sub:enumeration)). 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. [4.2](#fig:goanna-example-1) and Fig. [4.3](#fig:goanna-example-2), 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. [4.4](#fig:multi-error)).

|  |
| --- |
|  |

**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.

### 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 [4.3.4.1](#sub:grouping)) 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 all can be assessed separately.

|  |
| --- |
|  |

**Inspecting a defective Haskell Program (left) in relation to the error messages output by the standard GHC 9.0.2 (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’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.

For instance, in Fig. [4.5](#fig:grouping-ghc), 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. [4.6](#fig:grouping-goanna). These errors can be addressed collectively, thereby improving the efficiency of the programmer.

### Discovering Potential Causes

When a type error arises, Goanna-IDE shows a list of possible causes in the debugging panel. Each possible cause consists of one or more locations in the code that require modification to rectify the type error. Clicking on a possible cause activates it. The locations are highlighted in the text editor, as well as inlay type hints suggesting the suitable type expected for that code slice. In the debugging panel, the activated cause is outlined with a red icon, while others are marked with a blue icon.

The causes identified by Goanna are comprehensive. Goanna will take into account potential causes in expressions, pattern matchings, type annotations, and type class constraints. Consequently, programmers will generally find the real cause by exploring Goanna’s diagnosis. Unlike most Hindley-Milner  based type inference, Goanna does not show a bias towards the unification order, thereby avoiding the left-to-right bias .

Note that Goanna’s fixes are 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, however, offers fixes that encompass a complete set of changes necessary for a resolution.

### Assessing Likelihood of Causes

One challenge of Goanna’s “find all causes" approach is the number of ways an error can occur can sometimes become too large to be useful in practice. Goanna employs multiple techniques to intelligently sieve the list. For the remaining list, Goanna employs a few heuristics to rank their likelihood and inform programmers which causes they consider first. Goanna-IDE uses a star-based rating system to signal the “likelihood” of each cause. 3 stars indicating the most likely cause, 2 stars and 1 star follows.

### Type Hints

In addition to suggesting which part causes that type error, Goanna-IDE explains why this is inferred by using in-situ type hints on necessary terms. The type hints are displayed as inlay decorations on top of respective fragments of source code. These type hints provide enough information for programmers to understand the type inference, and Goanna will leave out the terms that are irrelevant to the type error. Goanna’s type hints are also dynamic to the selected cause. Programmers can observe how the inferred type of each term changes by changing the selected cause. Many modern programming tools use inlay type hints to support understanding, such as Haskell Language Server , most often, these tools will display all type hints or none. Unlike in Goanna, these tools do not provide alternative sets of type hints for programmers to compare.

### Cross-module type error debugging

When encountering a type error spanning across multiple modules, Goanna-IDE will group the potential causes indicated by their module and declaration block. Clicking on any possible cause location will focus the editor on the corresponding module (Fig. [4.7](#fig:goanna-cross-module)). Goanna is the first tool to introduce cross-module type debugging. The way Goanna presents cross-module type errors is analogous to how run-time errors are presented in most programming languages. When encountering a run-time error, most programming environments show a call stack containing multiple file paths, and programmers can choose which file to start investigating. Often, programmers choose to start from the file authored by themselves instead of library files. Goanna uses this mental model to group potential locations that cause a type error by module and definition blocks.

|  |
| --- |
|  |

**Debugging a cross-module error in Goanna.** In this error, potential defects may appear in either module A and B. Goanna suggests 3 potential causes and fixes: 1) Change the type annotation of x to [Maybe Int] (Top). 2) Change the y variable on line 4 of module A to an instance of Int. 3) Change both the elements in the list literal in module B (Bottom), hence affecting the type of y. Clicking on each potential cause in the debugging panel results in different highlights and type hints in the editor panel.

## Goanna Implementation

Goanna comprises 3 phases: constraint generation, MCS enumeration, and post-analysis. In the constraint generation phase, Goanna walks the abstract syntax tree and collects constraints. In the MCS enumeration phase, Goanna enumerates through all MCSes. Lastly, in the post-analysis phase, Goanna applies multiple optimization techniques to reduce the number of MCSes, group MCSes by common properties, and sort them based on heuristics.

### Haskell Coverage

Goanna supports a wide and growing range of the Haskell 2010 language syntax . At the time of writing, fully supported features include module import/export, qualified imports, import hiding, do notation, algebraic data types, newtypes, type synonyms, type classes, operator sectioning, and range expression. Goanna does not yet support type features enabled through language extensions. However, they are also on the roadmap. A detailed and updated feature coverage is publicly available.

### Constraint Generation

Goanna uses the abstract syntax tree of the original Haskell program and translates it into a constraint program by modeling how types are defined and used. Goanna does not restrict which constraint language and solver should be used. The only requirement is that Goanna needs to be able to assert whether a subset of the constraints is still feasible by calling a provided solve function during the MCS enumeration phase. In our implementation, we generate portable Prolog predicates . The solve function executes a predefined predicate type\_check/0 that tests all the generated predicates. 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.

For a simplified Haskell syntax shown in Fig. [4.8](#fig:translation).A, we generate a list of Prolog predicates in the language shown in Fig. [4.8](#fig:translation).B. We use 4 auxiliary functions during the constraint translation process (Fig. [4.8](#fig:translation).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 . To clarify, all parsed Haskell syntax are in blue. All generated Prolog syntax are in red.

Let 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. [4.8](#fig:translation).D) take 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. [4.8](#fig:translation).E) take , a Haskell expression node or type node, and a Prolog variable V as input and output a list of Prolog terms. These terms attempt to unify the inferred type of provided node to the provided Prolog variable V.

|  |
| --- |
|  |

Goanna’s Constraint Translation Rules (Simplified)

An example of such translation can be found in Fig. [4.9](#fig:translation-example). In the Haskell program (Fig. [4.9](#fig:translation-example).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.

|  |
| --- |
|  |

**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.

### MCS enumeration

After the constraint generation phase, Goanna obtains a list of constraints derived from the source code and is able to query the feasibility of any subset of the constraint system by calling the solve function. Using a known algorithm , Goanna then derives some useful subsets of the constraint system through MCS enumeration. We refer to the complete set of constraints as a constraint system . When we use the word subset without specifying the corresponding superset, it can be inferred as the subset of the constraint system . We list these subsets obtained from MCS enumeration and give their type-theoretic interpretation.

– A minimal unsatisfiable subset (MUS) of a constraint system is a subset such that is unsatisfiable and is satisfiable. An MUS can be seen as a minimal explanation of the constraint system’s infeasibility. MUSes have been used extensively, mostly in combination with programming slicing, as a means to explain type errors. A MUS of type system constraints reasoning chain connecting all evidence from one location of the conflict to another. Goanna uses the set of all MUSes to group related type errors.

– A minimal correction set (MCS) of a constraint system is a subset such that is satisfiable and is unsatisfiable. MCSes are so named due to the fact that their removal from can be seen to “correct” the infeasibility. In an ill-typed program, an MCS can be seen as the “cause" of a type error; the removal of C will result in the system being well-typed. Goanna uses MCS to represent potential causes of a type error. Each MCS contains the set of locations that need to be changed to fully resolve the type error.

– A maximal satisfiable subset (MSS) of a constraint system is a subset such that M is satisfiable and 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 an ill-typed program, an MSS can be seen as the resulting typing environment if a type error is fixed by excluding the MCS . Goanna uses an MSS to provide type hints for the program even when it is ill-typed.

|  |
| --- |
|  |

**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.

For the example in Fig. [4.10](#fig:enumeration-example), Goanna’s MCS 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 ), the type annotation alone in g (from ), and the function definition alone in g (from ).

### Post-Analysis

To improve the quality of error diagnoses, Goanna performs three types of post-analyses: type error grouping, cause ranking, and cause reduction.

#### Type Error Grouping

Type error grouping is a novel feature provided by Goanna. Conventionally, in a type error slicing approach, a type error is represented by a minimal unsatisfiable subset (MUS). With multiple MUSes available, we have the knowledge to be more precise about an ill-typed program. We propose a novel method of representing type errors that aligns more closely with their colloquial meaning.

Let denote the set of all Minimal Unsatisfiable Subsets (MUSes) and the set of all Minimal Correction Sets (MCSes). We define an undirected graph , where each vertex in corresponds to a minimal unsatisfiable subset , and the edges of connect pairs of MUSes and if their intersection is non-empty. The set of all connected components in represents the set of all type errors. For each , let , where is the set of vertices in . is the set of all constraints local to this type error. Define 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. [4.11](#fig:grouping-example).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.

|  |
| --- |
|  |

**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, fixing 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. [4.11](#fig:grouping-example).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 of 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 group, 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 errors, each having potential causes, will result in total causes. Dividing these causes into separate errors that align correctly with intuition is the most important technique to enhance Goanna’s error reporting.

#### Cause Ranking

In Goanna-IDE, when a list of potential causes is on display, Goanna-IDE also shows the top 3 “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:

|  |
| --- |
|  |

**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. [4.12](#fig:loc-count), 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 to encourage the cause whose fix will result in every surrounding term to be as concrete as possible.

|  |
| --- |
|  |

**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.

In the example in Fig. [4.13](#fig:specificity), 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 x having a less concrete type. Indeed, x can have any type if not limit what function to replace not with. Goanna prioritizes the first cause over the second.

##### Error span.

Goanna prioritizes the potential causes whose corresponding locations are clustered within fewer function definitions and lowers the likelihood of those whose corresponding locations being spread across multiple definitions or even multiple files.

#### Cause reduction

We employ three techniques to elevate the clarity of the suggestion list: reduction of constraint count, elimination of over-fitting resolutions (solutions that are technically correct but involve unnecessary modifications), and elimination of redundant fixes.

##### Minimize Constraint Count.

The number of constraints directly influences the time complexity associated with enumerating the Minimum Correction Subset (MCS). By merging multiple constraints into a singular one, we can reduce the total count of constraints and, in turn, improve the performance of enumeration. Yet, this approach requires careful application, as it could potentially lead to unsolvable situations or propose infeasible fixes, as the combined locations in the source code must either all contribute to the type error or none do. An effective application of this technique is to merge all constraints created by sub-expressions in a type signature.

|  |
| --- |
|  |

**Combine constraints in type signature.** For a simple Haskell program (top), without any optimization, Goanna generates 10 constraints (bottom left), indicated by 10 subgoals in the predicate. By combining the constraints in the type signature, Goanna produces 2 constraints (bottom right).

Consider the example in Fig. [4.14](#fig:combine-constraints). Without optimization, this code would spawn 10 constraints. However, applying this optimization can achieve an equivalent outcome with merely 2 constraints. Notably, this optimization forfeits the capability to suggest fixes for sub-expressions within a type signature, a decision that warrants some consideration but, in our experience, pays off.

##### Elimination of Over-Fitting Resolutions.

Over-fitting often refers to a situation where error correction tools provide solutions that are technically correct but involve making unnecessary modifications. In Goanna, every syntax node in the source code generates one or more constraints. This includes structural nodes such as function applications and let expressions. However, very often, suggesting that the user should modify the entire function application expression is not particularly instructive when changing one of the arguments fixes the type error as well. Disabling suggestions for over-fitting solutions improves the clarity of the suggestion list and enhances the speed of MCS enumeration.

##### Elimination of Redundant Causes.

Goanna iterates over the possible causes and removes the ones that fail to deliver new insights. If all locations in a cause suggestion have already been covered in preceding suggestions, subsequent suggestions that merely rearrange these locations in different combinations can be omitted.

|  |
| --- |
|  |

**The number of potential causes can grow exponentially.** In the ill-typed Haskell program (Top), there are 4 different ways (Bottom) to fix the type error. It is not hard to see this growth is exponential, and showing all the alternatives is not helpful.

Consider the example in Fig. [4.15](#fig:reduction-example). Without knowing the true intention of the programmer, Goanna can provide four ways to fix the issue shown at the bottom. However, closer inspection reveals that after the first two suggestions, we no longer unearth new insights. Therefore, they can be removed to enhance the clarity of the suggestion list. Note that this can remove the correct answer (say D), but if the programmer uses part of the (A) to make the fix, the revised type error will include the correct fix.

Removing superfluous MCS-based suggestions that recycle different permutations of the same set of locations is an instantiation of the Set Cover Problem (SCP). The problem can be rephrased as finding the minimal number of MCSes that cover all the potential locations that could cause the type error. Many approaches solve the SCP , including eager algorithms, linear programming, and heuristic-based algorithms. Generally, we found all of these approaches find the minimal cover of type errors efficiently. Goanna uses the OR-tools for this task.

## Evaluation

We want to answer the following key research questions about our Goanna prototype:

* RQ1. Does Goanna provide a more accurate type error diagnosis compared to traditional tools?
* RQ2. Does Goanna provide a concise list of suggestions?
* RQ3. How efficiently does Goanna compute error diagnoses?

### Experiment Design

To assess Goanna, we extracted a collection of defective Haskell programs (N=86) from Haskell online discourse since 2018, each containing one or more type errors. The communities we searched include StackOverflow (32), Haskell on Reddit (20), and Haskell Discord Channel (34), as these are the top discussion channels for Haskell users . During the search process, we looked for online discussions where the authors encountered type errors in their Haskell programs and asked for help. Further, we selected only the questions that had been answered and accepted by the original author. We extracted the defective Haskell programs and the accepted answers as the oracle solution. The length of these programs ranges from 1 liner to 64 lines of code (mean=20, median=20). These programs span a variety of subjects, 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). The distribution of themes generally aligns with the breakdown of the different causes of the type errors from Tirronen’s study .

For each metric in the evaluation, Goanna is compared with Glasgow Haskell Compiler (GHC) , and Helium . GHC was chosen as the baseline due to its established reputation in the Haskell community, wide capability, and great efficiency in working with Haskell projects. Helium is acknowledged for producing high-quality error messages . The experiments were run with GHC 9.4 and standalone Helium compiler version 1.8.

### RQ1. Accuracy

For each program in our dataset, we compared each tool’s error diagnosis with the accepted answer. We consider the diagnosis accurate only if its suggested fix matches the accepted answer. We consider the diagnosis partially accurate if the tool’s diagnosis is part of a larger set of locations that make up the intended cause or if the diagnosis addresses one of the multiple errors. Because Goanna provides a comprehensive list of possible causes, it is very likely all sensible fixes are included. In this evaluation, we only consider the top-1 suggestions (Goanna 1) and top-3 suggestions (Goanna 3). From the graph in Fig. [4.16](#fig:accuracy), GHC’s accuracy is the least performant among all tools. Goanna 1’s, although lower than Helium (72.1%) in partial accuracy, is higher than Helium (51.2%) when considering only diagnoses that fully match the accepted answer. **Goanna 3 has the best accuracy (84.8%)**, higher than the partial accuracy of other tools.

|  |
| --- |
|  |

The percentage of diagnoses that match the accepted answers.

### RQ2. Conciseness

Using Goanna requires users to cherry-pick from a list of possible causes. It will severely reduce the usability if the list is too long. To evaluate Goanna’s conciseness, we counted the number of suggestions provided by Goanna in all the tasks. We also indicate where the accepted answer is. Additionally, we included a baseline of Goanna with all the cause reduction features disabled. As shown in Fig. [4.17](#fig:conciseness), **Goanna manages to effectively condense its suggestion list**, on average providing a short list of suggestions (mean=3.29, median=3.0) for each type error. Additionally, on average, the accurate cause can be found within the top 2 suggestions (mean=1.63, median=1.0) to find the correct cause identification and fix. It also shows in Fig. [4.17](#fig:conciseness) that Goanna’s cause reduction strategies are effective; on average 51% of total causes are reduced to gain clarity. One unusual observation is that in 3 tasks, Goanna failed to include the correct solution. The current version of Goanna is ineffective in making the correct suggestions for these type errors. We discuss this in Section [4.5.2.0.2](#sec:edge-case).

|  |
| --- |
|  |

The number of potential causes identified by Goanna.

### RQ3. Performance

Goanna’s performance largely depends on the MCS enumeration phase. Enumerating all MCS is computationally expensive. We experimentally compared the time it takes for Goanna to provide a complete error diagnosis for each task with GHC and Helium. From the data shown in Fig. [4.18](#fig:performance), we can see that **Goanna is slightly slower than Helium** (Goanna: mean=0.98 seconds, median=0.83 seconds; Helium: mean=0.63 seconds, median=0.63 seconds). Goanna is about 10 times slower than GHC (mean=0.09 seconds, median=0.09 seconds), but greatly outperforms GHC (see Fig. [4.16](#fig:accuracy)). One important pattern is that Goanna’s response time varies more than other tools (Goanna SD = 0.55, GHC SD = 0.00, Helium SD = 0.10). It can be seen from Fig. [4.17](#fig:conciseness) and Fig. [4.18](#fig:performance) that the tasks Goanna struggles most with are the ones that have significantly more potential causes. Multiple avenues exist to mitigate this delay, and we discuss them in Section [4.5.4](#sec:future-work).

|  |
| --- |
|  |

The time it takes to type check and diagnose each program.

### Threats to Validity

##### **Selection of the dataset**

Our selection of dataset is limited in its number. This is due to the challenge of finding programs that contain type errors. Unlike runtime errors, which can be mined from code repositories and version control histories, type errors in Haskell can be detected by the compiler tool, and ill-typed programs are usually fixed before the changes are committed to the version control systems. Further, we employed two selection methods. First, the error is indeed a type error. We test this by running the original program in GHC and checking if it indeed triggers a type error. The program is discarded otherwise, for instance, if it contains only parsing errors or runtime errors. Second, we discarded type questions where the main error relies on third-party libraries.

##### **Measurement of performance**

Performance on Goanna and GHC was measured on a Linux virtual machine with a 3.1 GHz Processor and 2GB RAM. In practice, complex systems like this may perform differently depending on hardware and software configurations. Although we were not able to extract the performance profile of each tool across different platforms and operating systems, we chose hardware with abundant resources and up-to-date software dependencies. During our performance measuring, neither CPU usage nor memory usage was fully stressed. Additionally, GHC was run with the “-fno-code” flag enabled to limit its usage to type-check only.

## Discussion

Drawing on the evaluation’s findings, we identify Goanna’s strengths and weaknesses. We propose directions for future work to provide useful applications and address its current limitations.

### Strengths

Goanna demonstrates notable improvement over existing Haskell type error detection and repair tools. Compared with traditional type-checking tools such as GHC, Goanna delivers improved error detection accuracy and flexibility to inspect different potential causes. The data suggest that users typically need to consider only 2-3 suggestions to achieve a satisfactory result. It improves error localization organization, avoiding the presentation of all error locations at once.

##### **Accurate suggestions**

Goanna is able to identify causes for Haskell type errors more accurately than GHC and Helium. We attribute this to a few factors. First, Goanna is the only tool capable of suggesting the type error in more than one node. In a study on how students fix Haskell type errors , of over 2700 ill-typed Haskell programs, only 35% of the type errors were caused by a single location. However, most of the type debugging tools only focus on single-location causes due to their technical limitation or to avoid high computational cost. Second, Goanna is the only tool that provides alternative causes of a type error. We were able to see that although Goanna-1 is not as accurate as Helium when accepting partial fixes, Goanna-3 surpasses Helium in accuracy. This translates into accurate type error identification at the cost of presenting the top 3 answers from Goanna instead of one.

##### **Goanna provides contextual information**

Goanna provides type information for relevant terms to support each of its claimed causes. In traditional tools, the type-level information is often incomplete or totally discarded. In runtime error debugging, one of the most common features is inspecting the values of different expressions of the program. It would be ineffective if the runtime debugger only showed the location of the error. Goanna simulates this feature in the type debugging setting. Instead of run-time values, Goanna allows programmers to inspect the type assignments and observe how they change with different assumptions of the potential cause of a type error.

### Weaknesses

The current implementation of Goanna also has a few drawbacks. We provide two: its moderate performance and its tendency to misunderstand the type of error in some edge cases.

##### **Responsiveness**

The trade-off of extensive analysis undertaken in Goanna results in a substantial delay (mean=0.98s). Goanna’s current performance is not yet suited to real-time feedback in programming tasks. Based on Nielsen’s suggestion for wait time tolerance , Goanna should provide responsive answers ( 1 second) for real-time programming analysis, where users’ flow of thoughts stays uninterrupted. Even in larger and more complex tasks, Goanna’s response time ( 10 seconds) is still suitable as an on-demand tool when a complex type error occurs. As shown in Wu’s study , in the simplest situation where students fix the type error in a single step, it will usually take about 60 seconds to complete the task. This error resolution time grows in proportion to the number of steps the students take to fix the error. If Goanna is able to shorten the steps to final resolution, then the querying time will easily be offset by the time it saves.

##### **Edge cases**

Although Goanna’s error reporting is, in general, exhaustive, there are situations where Goanna still fails to provide insightful diagnosis. One general theme is that Goanna is very effective when the error requires modifying a syntax node but can be less insightful when the intended fix is to insert, delete, or rearrange syntax nodes. In the example in Fig. [4.19](#fig:weakness), Goanna suggests changing the map function to a function of type [Int] -> [Bool]. But in practice, a human user would very easily identify that an expression defining the function to be mapped is missing between the map and the list being operated on.

|  |
| --- |
|  |

In this type error, Goanna suggests changing map to a function of type [Int] -> Bool. Although this is technically correct, in practice, a human expert user would easily identify that a function expression is missing between the function map and the list literal.

### Multi-witness and Multi-party Type Errors

Although not supporting granular tracing of multi-step type errors as ChameleonIDE [3](#chap:chameleon) does, Goanna is especially well-equipped for investigating multi-witness and multi-party type errors. This advantage comes from Goanna’s ability to obtain all MUSes. As we discussed in Chapters [2](#chap:haskell-type-checking), multi-witness and multi-party type errors all involve more than one MUS. For multi-witness type errors, Goanna correctly identifies all the relevant witnesses. In comparison, tools based on a single MUS, such as ChameleonIDE, pick one witness from each endpoint because of the minimality of MUS. In addition, Goanna is also able to divide multi-party type errors into multiple simpler ones using the type error grouping feature, providing further clarity into the debugging workflows.

### Future Work

It is important to evaluate Goanna with human participants and gain qualitative insight into its effectiveness. In one of our preliminary workshop studies, participants showed positive reactions after using Goanna. A rigorous human study based on realistic debugging use cases is planned in the near future.

Several areas of potential enhancement could improve Goanna’s functionality and efficiency. One exciting path of improvement is to generate suggestions for syntax changes on top of type changes. As pointed out by Chen and Erwig , syntax changes are much more challenging than type changes. But with the recent improvements of generative models and research advancement in ML-based type error , accurate syntax changes may be on the path to becoming feasible.

Several approaches to achieve higher performance of Goanna show promise. The parallel capability of state-of-the-art MUS enumeration algorithms are not explored in this study. With proper implementation, it will be possible to lower the hardware barrier of entry for wide adoption of MCS-based type error suggestion tools. Further, with domain-agnostic MUS enumeration tools , it should be possible to consistently achieve high performance while using a more performant constraint system and proper parameterization. Lastly, in a real-world implementation, it is possible to employ partial MUS enumeration to restrict the enumeration process in a sensible time-bound.

Additional future work could include examining Goanna’s integration with other tools within the Haskell development ecosystem. For instance, including Goanna in widely used text editors or development environments could offer developers a more integrated and fluid experience and be an important avenue for Goanna to reach a wider audience.

Finally, it may be important to consider extending Goanna’s capabilities to support functional programming languages other than Haskell. Several other languages, including Scala and OCaml, also support static typing and type inference, and Goanna could be a valuable addition in these contexts. In the future, we hope to extend these techniques to popular multi-paradigm languages such as TypeScript and Rust.

## Related Work

In this section, we survey the corpus of work that Goanna built upon and was inspired by. We first review the work that is built on a theoretical foundation similar to that of Goanna. We then examine studies that share similar aims and offer a debugging experience comparable to Goanna’s. Finally, we delve into research on program slicing and discuss how Goanna extends the concept of slicing and distinguishing itself from its predecessors.

### MCS enumeration

MCS enumeration has been extensively studied in the field of error localization across various domains. In the specific context of Haskell type error diagnosis and resolution, several related approaches have been explored. It is important to examine the strengths and weaknesses of Goanna in the context of these areas.

One notable work  introduced a tool that utilizes the capability of MCS to localize multiple faults and identify software defects using unit tests. Their approach demonstrated the effectiveness of MCS in pinpointing errors within a program. Similarly, Bekkouche conducted a relevant study  on MCS utilization for locating program errors in while-loop programs. Their findings showed improved efficiency compared to SAT-based approaches. Although showing strength in programming language static analysis, MCS-based fault localization has not been previously applied at the type system level. Goanna distinguishes itself as the first tool to explore this approach within the realm of type error diagnosis and resolution.

### Suggesting changes to type errors

Seminal  uses syntax mutation and binary search to find appropriate syntax changes to program errors. The advantage of Seminal is that it’s capable of suggesting direct syntax changes to common mistakes (e.g., mistakingly swapping the order of function arguments). However, it is impossible for Seminal to provide the complete set of all potential fixes. Nor does it guarantee a suggested solution is minimal syntax change.

Counter-factual typing (CFT) uses a variation-based type system; it is capable of suggesting the correct type for all possible. CFT shares many capabilities with Goanna, CFT is able to suggest multiple-location changes, CFT uses similar ranking heuristics. Goanna is able to produce an in-depth analysis of the ill-type program, such as type error isolation. CFT and Goanna both aim to produce a complete set of potential fixes; Goanna employs a set of effective algorithms to reduce the exponential number of potential fixes without reducing the quality of suggestions.

SHErrLoc uses constraints as the underlying to perform type inference and type error diagnoses. SHErrLoc is able to suggest multiple possible fixes of the type error and rank them based on heuristics. Unlike Goanna’s approach of using a general-purpose constraint language, SHErrLoc relies on GHC’s internal constraints and then translates them into SCL (a custom-made constraint language). On the technical side, this approach relies heavily on modification of the compiler and does not remain reliable with later GHCs. Most importantly, there is no way to interact directly with the solver. This renders the kind of constraint manipulation in Goanna and SHErrLoc impossible. Goanna is able to perform type reconstruction for ill-typed programs, that is, finding the most concrete types for all expressions for each potential solution using the Maximal Satisfiable Subsets. SHErrLoc focuses on finding the locations only.

### Type Error Slicing

Type error slicing is a technique to identify all necessary locations of a type error that is necessary for programmers to diagnose the root cause. It has been studied in many studies ever since . These studies all use Minimal Unsatisfiable Subset (MUS) to ensure the *completeness* and *minimality*. The drawback of type error slicing is that it often produces too many locations. Chameleon improved type error slicing by allowing programmers to interactively show the partial MUS by choosing their own assumptions. Compared to these tools that base their analysis on a single MUS, Goanna effectively utilizes all possible MUSes. One benefit of enumerating all MUSes is Goanna covers all locations of multi-witness type errors, while Chameleon reports partial locations. Another benefit is the availability of all MUSes lends Goanna a more accurate representation of multi-witness type errors (all witnesses are identified) and multi-party type errors (complex type errors are untangled into simpler ones). These benefits are, however, at the cost of the computational complexity in MUS enumeration.

## Summary

In this chapter, we introduced Goanna, a tool for identifying and resolving type errors in Haskell code. We described the features of Goanna, including its fix suggestions, type error grouping, and identifying multiple type errors. We also discussed our approaches to reduce and reprioritize fix suggestions while maintaining comprehensiveness. Additionally, we walked through the uses of Goanna-IDE, a type error debugging interface for Haskell.

We evaluated the effectiveness of Goanna from a set of 86 diverse Haskell programs and demonstrated its ability to identify and resolve type errors accurately compared to other tools. We also showed that Goanna effectively condenses the list of causes. When a large number of causes is inevitable, Goanna’s suggestion ranking heuristics ensures that more useful fixes are prioritized. Goanna currently works with Haskell, but in the future, we plan to extend its MCS-based error diagnosis to work with other strongly typed languages, such as Rust and TypeScript.

# GeckoGraph: A Diagrammatic Notation for Haskell Types

This chapter introduces GeckoGraph, a visual language designed to support easy inspection and comparison of polymorphic types. We begin by highlighting the challenges associated with understanding and using polymorphic types. Next, we introduce GeckoGraph, demonstrating how it addresses these challenges and outlining the process of transforming text-based type signatures into GeckoGraph. We then present our experiment, evaluating the effectiveness of GeckoGraph in assisting with program tasks involving polymorphic types. Finally, we discuss the strengths, limitations, and potential applications of GeckoGraph.

This chapter uses the content from our paper *GeckoGraph: A Visual Language for Polymorphic Types* , with slight modifications to ensure coherence within this thesis. This work was publicly available as a preprint when submitting this thesis.

## Introduction

In programming languages, a polymorphic type can represent values of different types while providing a common interface or behavior for those values. Polymorphic types are central to the succinctness of contemporary statically typed functional languages, enabling a considerable degree of type-safe abstraction and, hence, component reuse. Parametric polymorphism is available in many programming languages, from functional languages such as Haskell and ML to imperative and multi-paradigm languages such as Rust  and Go . Polymorphism allows programs to be written in a way that is more generic and adaptable to different data types, enabling greater flexibility and code reuse. Polymorphic types are ideal for modeling abstractions, such as properties of mathematical objects and laws that hold on these objects.

Although polymorphic typing promises robustness and a high degree of code reusability, studies  show that using polymorphism in practice poses challenges, especially for novice users. These studies have shown that humans tend to focus on concrete types and only rely on polymorphic type checking as a last resort. In practice, polymorphic types often pose usability problems for programmers. New polymorphic type variables can be created during type checking. These intermediate type variables (such as t0 and a0 in Fig. [5.1](#fig:example-foldable)) are often kept behind the curtain unless a type error is encountered. This often results in programmers resolving type errors with type variables not authored by the programmers themselves (Fig. [5.1](#fig:example-foldable)).

|  |
| --- |
|  |

An example type error where a programmer mistakingly provided a Char instead of a String literal. **Left –** The compiler (GHC 9.0.2) shows an error message comparing the provided Char type to a confusingly named type t0 a0. **Right –** GeckoGraph shows the exact message with the two types in graphic notation, highlighting the structural difference rather than identifier names.

While type polymorphism is one of the oldest topics in programming language theory , little research focuses on the *usability* of polymorphic types. Hage argues that the expressiveness and power of type systems often come at the cost of usability . We aim to investigate the challenges of using polymorphic types and explore how to improve their usability with visualization and modern HCI techniques. To achieve this, we propose GeckoGraph, a graphical notation for types. GeckoGraph aims to complement traditional text-based type notation and make reading, understanding, and comparing types easier. GeckoGraph is prototyped and verified iteratively, leading to a design with visual clarity applicable to many programming contexts. Our study evaluating 714 participants’ ability to solve type adaptation challenges in GeckoGraph versus text-based type annotation is, to our knowledge, the largest controlled user study of a functional programming tool or concept. The study results show that GeckoGraph helps improve programmers’ ability to succeed in resolving programming challenges, especially for novice programmers.

## GeckoGraph

GeckoGraph is a visual notation for type annotations in statically typed programming languages. It is intended to work tangibly with text-based annotations, but uses colors, shapes, and symbols to make structures of types easy to identify at a glance. In this section, we describe the design of GeckoGraph and highlight some unique benefits of programming with GeckoGraph.

### Design of GeckoGraph

The design of GeckoGraph focuses on visualizing types in functional languages (e.g., Haskell, ML). We provide a working implementation of GeckoGraph for Haskell. As illustrated in this section, it can express basic types, polymorphic types, algebraic data types, and some advanced type-level features. However, GeckoGraph could also be used in imperative and multiparadigm languages such as Rust. We provide a GeckoGraph construction library for Haskell.

We identified three main design goals for GeckoGraph based on the challenges of using polymorphic types  and how programmers tend to use type annotations , as follows.

##### **(D1) Low barrier to learn**

GeckoGraph should take little to no effort to learn. The rules for translating a text-based type notation to GeckoGraph should be minimal. Where possible, GeckoGraph should be intuitive to programmers who are familiar with text-based type notation.

##### **(D2) Easy to parse for humans**

GeckoGraph should make the task of reading and understanding type notation easy. It should emphasize the less obvious properties of a type signature. GeckoGraph should eliminate the need for mental backtracking, such as counting opening and closing parentheses and remembering which type classes are required on which variables.

##### **(D3) Easy to compare and search**

GeckoGraph should aim to make the task of comparing two types easy, especially to make subtle differences in text-based notation harder to miss. This also includes the task of choosing an ideal function from a list of potential functions. For example, programmers search for a desired function from a documentation site with only partial knowledge of its type (e.g., the arity, one of the argument types, or type class it must fulfill).

##### Simple Types

Simple types, such as type variables and concrete types, are displayed in a cell image: a solid-colored rectangular box with an angled corner on the top left. Each type identifier encodes a distinct color hue of the cell. Its first 1 or 2 letters are displayed inside the cell at the bottom left to provide familiarity (design goal [5.2.1.0.1](#goal1)) and strong secondary encoding. The angled corner in the top left provides visual separation between two cells, even when the same color cells are next to each other, allowing GeckoGraph to be zoomed out to extremely small sizes (Section [5.5.2.1](#subsec:space)) without suffering readability (design goal [5.2.1.0.2](#goal2)).

|  |
| --- |
|  |

Examples of how various types are represented in GeckoGraph, including type variables (A) and concrete types (B). Data types (C, D, E), function types (F, G, H), and type classes (I, J).

##### Data types

GeckoGraph displays an algebraic data type as a larger cell, where the type constructor half encloses its arguments: . The arguments are aligned in the bottom right of the cell. Two distinct visual dimensions are used to provide additional visual clarity (design goal [5.2.1.0.2](#goal2)). Data types containing more arguments (e.g., a b c or (a b) c) will expand horizontally . Data types that are nested (e.g. a (b c)) will grow taller . This distinction accommodates our design goal [5.2.1.0.3](#goal3). Note that the height of GeckoGraph grows only upwards, but not downwards. Not only does this allow GeckoGraph to be more efficient in its space usage, but it also allows the legend text to be correctly aligned at the bottom and can be read similarly to regular type notation (design goal [5.2.1.0.1](#goal1)).

##### Function Types

Functions are the fundamental building blocks of functional programming languages, and function types are ubiquitous and the most important in type-level programming. In Haskell, (->) is defined as an infix type operator with right associativity to provide a succinct type annotation. GeckoGraph preserves this syntax feature to make the notation more intuitive (design goal [5.2.1.0.1](#goal1)): the 2 arguments of a function type in GeckoGraph are placed on both sides of the cell . A special function indicator () is displayed at the top of the cell.

Curried functions (e.g., a -> b -> c) display as two cells of functions merged together ; the second function overlaps on top of the first, indicating that the second function is the return type of the first. Regular higher-order functions (e.g., (a -> b) -> c ) follow the rules of functions and nested data types . The placement of function indicators aims to make it easy to find desired functions in the documentation site based on function arity and higher-order functions (design goal [5.2.1.0.3](#goal3)). It is easy to tell higher-order functions from the vertical position of its function indicator. Similarly, it is easy to count the arity of a function by counting the number of horizontally connected function indicators (Fig. [5.3](#fig:indicator)).

|  |
| --- |
|  |

An example of using the function indicator. The function indicator can be used to easily identify the arity of a function type by counting the connecting function indicators. For higher-order functions where functions are arguments of other functions, it is very easy to see the “order" of functions and how they are arranged.

##### Type Classes

Type classes are an intrinsic part of Haskell , and many other functional languages. In GeckoGraph, the type classes (e.g., (A a, B a) => a) are indicated in the extended area below one or more GeckoGraph cells . Each type class required on a type variable displays as a square indicator aligned on the right of the extended area. In GeckoGraph, type-class constraints are associated with every instance of the type variable that requires them. This means when displaying the type (==) :: Eq a => a -> a -> Bool in GeckoGraph, the constraint Eq appears in both occurrences of a.

The GeckoGraph type class’s design promotes the type class placements rather than the type class names. Programmers can easily see where and how many type classes are required, but they may need an extra step (global legends of the color mapping or pop-up window) to identify the name of the type class. We believe that this decision is well justified. For example, when reading a type (A a, A c, B a, B b, C b) => a -> b -> c, programmers may need to switch back and forth to remember which type classes are needed on which variable. GeckoGraph helps minimize the effort to associate each type variable with all its type classes (design goal [5.2.1.0.2](#goal2)).

### Benefits of Using GeckoGraph

Based on our design principles of GeckoGraph and our proposed glyphs, we outlined a few potential benefits of GeckoGraph over text-based type annotations.

##### Generalization patterns in type classes

A frequently cited confusion among novices is the symmetry of Foldable a and [a]. The subtle differences are often not important for beginners. However, when encountering type errors in working with lists, Haskell often explains the error with the Foldable instance. For the example in Fig. [5.1](#fig:example-foldable) where a programmer mistakingly provided a Char instead of a String literal, the compiler shows an error message comparing the provided Char type with a confusingly named type t0 a0. Although any Foldable instance is perfectly suited for the list , this generalization may reduce programmers’ confidence in their understanding of the language and the ability to navigate out of a type error. In GeckoGraph, a list type and a type with a Foldable instance have the same shape. This allows the generic type t0 a0 in the error message to assume the same shape as [a]. This generalization of concrete types and abstraction of type-class instances aims to allow for teaching fundamental functional programming concepts without hiding high-level abstractions. The same benefits apply to polymorphic numbers and strings.

##### Consistent color scheme

A common task in programming is to scan for a desired function from a sea of potentially useful functions, such as library documentation. During scanning, programmers often have partial knowledge of the desired function, e.g., the arity, one of the argument types, or the type class it must fulfill. A typical example is conversion: using a known String type to produce a desired Data.Text type. Another example is the ‘lookup’ function: using a known Data.Map a b to produce a desired b type. GeckoGraph supports this task by using consistent colors for the same type identifier. Programmers can rely on the color grouping to scan for the desired type from their own source programs or from third-party library documentation.

##### Advanced Type Feature Visualization

The design of GeckoGraph enables the visualization of many advanced type-level features. **Kind visualization**: if the *kind* of type variables can be inferred, the *kind* information is consistently displayed in GeckoGraph. For example, in Fig. [5.4](#fig:advanced-features) (A), the variable a needs at least the *kind* \* -> \* because of its use on the right-hand side. GeckoGraph respects this *kind* information and displays it as a constructor type over an empty structure, indicated using a dotted outline. **Qualified constraints**: GeckoGraph’s type class notation naturally extends to support qualified constraints. In the type forall b. A (a b) => a b, GeckoGraph shows the scope type class requirement on a b (Fig. [5.4](#fig:advanced-features) B). **Multiple Parameter Type Class**: GeckoGraph supports multiple parameter type classes by using multiple shapes with the same color hue to indicate the different parameters of the same type class. For example, for the type A a b => a b, GeckoGraph shows that the variables a and b both need an A class, but they are the different parameters of A (Fig. [5.4](#fig:advanced-features) C).

##### Precise Interactivity

Modern programming environments often allow programmers to mouse over part of the source code to query detailed information, such as definitions, references or documentation. However, with text-based source code, it is often hard to distinguish whether programmers want the most specific fragment under the cursor or larger blocks. Because of its graphical layout, GeckoGraph allows programmers to precisely select which part of a type signature they intend to query, that is, in Fig. [5.4](#fig:advanced-features) (D) when the user mouses over the type class box (orange square) under the second occurrence of a the type class it represents is revealed in detail.

|  |
| --- |
|  |

Advanced features of GeckoGrap. (A) GeckoGraph supports Kind Visualization if the inferred kind is greater than \*. (B) GeckoGraph supports qualified constraints by extending the extended area across multiple type variables. (C) GeckoGraph supports Multiple Parameter Type Classes, using different shapes of the same color to indicate that multiple variables must satisfy certain type classes collectively. (D) GeckoGraph supports the precise selection of its sub-structures.

##### Language Agnostic

GeckoGraph can be implemented in any language that uses static typing. In programming projects, GeckoGraph supports polyglot programming projects. Typical circumstances include projects using foreign function interfaces or multiple languages for client- and server-side programming. GeckoGraph provides a common notation to describe the functionality and features of systems. In teaching and learning programming languages, GeckoGraph removes the nomenclature difference in different programming languages. For example, when describing algebraic data types, different language communities use various names: tuple, enum, struct, etc. It is important to realize that these are the same concepts and ignore the minute linguistic barriers.

### Previous iterations of GeckoGraph

GeckoGraph was designed through many different iterations. Many research methods were used to verify ideas, including prototyping, cognitive walk-throughs, and formative studies. We list some notable design elements and their major feedbacks.

|  |
| --- |
|  |

Previous Versions of GeckoGraph. Different encodings represent named types, type variables, type constructors, and higher-order functions.

##### **Encoding the direction of function types**

This can be found in the design v1 (Fig. [5.5](#fig:previous)). This design shares many similarities with a prior project in type visualization. Although many students and experts agree on this, its use of space on both width and height scales is proportional to the types in question, causing too much inconvenience. In addition, this design does not produce a canonical form for a curried function. For example, for composition (.), it is unclear whether it takes two functions as input and returns a binary function or two functions and a single structure as input and returns a different structure.

##### Encoding the depth and size

Grayscales, gradience, size, and depth are promising visual representations for depicting numeric dimensions such as the depth and width of the parse tree. However, gradient and elevation were dismissed because of their requirements for a more demanding rendering process, making GeckoGraph harder to implement in more restricted user interfaces, such as the command line. In addition, all of these visual dimensions reduce readability when scaled down to a very small size.

##### Encoding symbolic names

In some variations, we tested using different geometric shapes to indicate the symbolic names of type variables and concrete types. We decided against using icons due to the limited number of different shapes until they were indistinguishable. The color provides more encoding spaces, and the letters provide familiarity with the original type annotations. This was shown to help reduce friction in the adoption of GeckoGraph.

## Evaluation

To evaluate the usefulness of GeckoGraph, we designed a controlled experiment in the form of a game called “Zero to Hero”. The game contains 10 levels of varying difficulty. At each level, participants are asked to implement a function called "zeroToHero" using only a list of available functions. These available functions are different at each level, and the target types of Zero and Hero vary at each level. The details of each level are provided in the Appendix (Appendix [7](#levels)).

The experiment aims to study how polymorphic types are used and reasoned about during programming tasks. In particular, we studied how programmers scan and select potentially useful functions from a library and compare intended types and actual types during type errors.

### An example level

We illustrate the task of the user study using level 4 of the game. At this level (Fig. [5.6](#fig:level-example)), the programmers are tasked with implementing the function zeroToHero :: Zero a b -> Hero b b. The available functions are f1::Zero a b -> Hero b a, f2::Zero a a -> Hero a a, f3::Zero a b -> Hero b a, and f4::Zero a b -> Hero b b. Two generic functions ($) and (.) are provided to improve the ergonomics of composing functions, but all tasks can be won without the use of generic functions. The possible solution and other details of the level can be found in the appendix (Appendix [7](#levels)).

|  |
| --- |
|  |

A screenshot from the game ZeroToHero. On this level – level 4 (Shown at F) – the players need to implement the function zeroToHero :: Zero a b -> Hero b b (B). They write their own definitions in the code editor (D) using a set of provided functions (G). The inferred type of their current definition is shown in (C). When ready, they can test their solution by clicking on the *Attempt* button (A). They can also skip a level by clicking on the *Bypass* button next to it. The output from the compiler, if there is any, is shown in a window below (E). The GeckoGraph in the screenshot uses a different color scheme that is optimized for computer screens.

At each level of the game, the programmer must select the right functions to achieve the target result. In particular, for this level, participants must discover that only f4 and f2 are necessary to produce the desired results. An implement that satisfies the target type is zeroToHero z = f2 (f4 z).

### Recruitment

Participants were recruited online through the Haskell community on Reddit and Discord. Participation was fully anonymized; detailed ethical implications of these experiments were reviewed and approved by the IRB of the authors’ institution.

### Group Assignments

The experiment uses a between-subject design. However, all participants receive both treatments (with and without GeckoGraph) during their runs. Participants are assigned to one of two groups. Both groups receive the same tasks in the same order. Participants in group one are assisted by both the text-based type annotation and GeckoGraph on even levels and only text-based type annotation on odd levels. Group two participants receive the same tool assistance but with the order flipped. The number of participants in the two groups is counterbalanced.

### Hypothesis

In programming tasks that involve reading and understanding polymorphic types, graphic notation using visual elements that provide high grouping strength (colors, shapes, sizes, and symbols) can improve the performance of such tasks compared to traditional text-based type notation. Our null hypothesis is that "Using graphic notation has no effect compared to traditional text-based type notation." This hypothesis and the task design were registered at the Open Science Foundation prior to data collection.

### Task Design

Participants in both groups receive the same 10 tasks. The tasks start off easy but gradually increase in difficulty. In each task, a target type signature of the function zeroToHero is given to the participants. Participants are provided with a list of available functions to implement the target function. This is to simulate the tasks of selecting useful functions from a library. In addition, participants are not allowed to use any other functions or variables outside the provided functions; even the Haskell prelude is not available. This ensures that everyone has the same knowledge and minimizes the effect of familiarity.

Participants can skip a level during the game if they are stuck. We believe that it is normal for anyone to get stuck on a challenging task, and being stuck on one of the 10 tasks does not discount their qualitative input of the tool. We limit 4 skips that a participant can use throughout the game, so that submitting qualitative feedback without completing at least some levels is impossible.

### Measurements

During the study, the time spent by participants on each task is recorded. We also record the resulting status of each level, whether it is a success or failure. Before each run, participants nominate their level of Haskell experience on a four-level scale: beginner, familiar, knowledgeable, and expert. If a participant has completed all 10 levels (with the help of skipping), we invite the participant to complete a post-study survey. In it, we ask for their opinion on how intuitive the GeckoGraph design is, how distracting they find GeckoGraph, and how helpful GeckoGraph is during the game, using a seven-point scale. In the end, we asked a few open-ended questions, inviting participants to provide their experience using GeckoGraph and their expectations about its potential applications.

Data collection from the human study was stopped after the planned cut-off period of 14 days. After the cut-off date, the ZeroToHero game is open source and available for free evaluation and repeating our experiment, but no further data was collected.

## Results

During the data collection period, 714 users participated in the study. Of these, 245 were novice users, 216 were familiar with Haskell, 216 were knowledgeable users, and 88 were expert users.

### Time to complete levels

The 10 levels are designed to gradually increase difficulty. Based on the results of the experiment, most of the tasks align with this trend. However, three tasks stand out in Fig. [5.7](#fig:level-time). Level 7 (mean = 334 seconds) is the hardest task in the game in terms of time, followed by level 8 (mean = 228 seconds) and level 5 (mean = 224 seconds). To complete an average level, the beginner group uses an average of 100 seconds, the familiar group uses 90 seconds, the knowledgeable group uses 80 seconds, and the expert group uses 70 seconds. This roughly aligns with self-reported expertise. We show that the task time on each level follows normal distributions using a Shapiro-Wilk test (p-value , for an alpha value of 0.05, p less than 0.05 is considered normal distribution).

Levels 5, 7, and 8 are the only three levels that include functions from the standard Haskell library, baring the (.) and ($) provided for convenience. Level 3 requires programmers to use the fst and snd functions to extract value from a tuple. Level 7 requires programmers to use the (<\*>) function of the Applicative class, while level 8 the fmap function of the Functor class. The authors speculate that the more experienced participants are much more familiar with these functions. This explains the strong contrast between these three levels.

However, when comparing the task time between the two treatments, we were unable to reject the null hypothesis. In a two-sample T-test, we could not find any significant differences between the two groups overall (p-value = 0.457). In addition, we could not identify a significant difference in each of the four experience groups either (beginner: p-value = 0.845, familiar: p-value = 0.524, Knowledgeable: p-value = 0.712, expert p-value = 0.771), an indicator that GeckoGraph has little impact (positive or negative) on time consumption.

|  |
| --- |
|  |

Time spent on each level, with 95% confidence interval. We show that the difficulty steadily increases across the game, but levels 5, 7, and 8 are significantly harder than the authors intended. The overall task time of each group roughly matches experience level.

### Success rate

We saw that, overall, GeckoGraph provides a higher success rate (96.88%) than text-based type notation (94.62%). This trend can be seen in every experienced group: beginner group (95. 12% vs. 92. 68%), familiar group (97. 39% vs. 93. 34%), knowledgeable group (96. 82% vs. 96. 06%) and expert group (98. 2% vs. 96. 40%). We saw the significance decrease as the user’s experience increased. When performing a proportion test on each group, we see that the effect is most significant with the beginner group and reject the null hypothesis (z score = 2.0228, p-value = 0.0431), followed by the familiar group (z score = 1.7495, p-value = 0.0802). The knowledgeable group (z score = 1.0295, p-value = 0.3032) and the expert group (z score = 0.8660, p-value = 0.3756) show less significant differences between treatments.

When breaking down the result in each task (Fig. [5.8](#fig:success-rate)), we were able to reject the null hypothesis in task 10 of the beginner group and task 10 of the familiar group [5.8](#fig:success-rate). We address this correlation in Section [[sec:gecko-discussion]](#sec:gecko-discussion).

|  |
| --- |
|  |

Success rate of each task with and without GeckoGraph, grouped by experience level. The figure is cropped from 70% to 100% for readability. In most tasks, GeckoGraph provides a small edge. However, significant differences were found in task 10 of the beginner group and task 10 of the familiar group.

### Qualitative Feedback

In their responses to the post-study survey, most programmers believe that the design of GeckoGraph is intuitive and that its appearance in the interface does not cause distraction. For the question “Do you find the GeckoGraph distracting?", most of the participants rated a negative score, with an average of 2.88 (Fig. [5.9](#fig:qualitative) left). For the question “How intuitive do you find the GeckoGraph?", we saw a reverse correlation of experience (Fig. [5.9](#fig:qualitative) Middle): experts find the GeckoGraph most intuitive (5.07), followed by the knowledgeable group (4.87), and the familiar group (4.80). The beginner group found it to be the least intuitive but still rated a positive score of (4.71). When answering the question “How helpful do you find GeckoGraph in finding the solution during the game?", the answer is more divided into different experience groups (Fig. [5.9](#fig:qualitative) right). It is slightly positive for beginners (4.25) and slightly negative for the other groups, the familiar group (3.86) and the knowledgeable group (3.32). The expert group finds that GeckoGraph is relatively unhelpful, with an average score of 2.95.

|  |
| --- |
|  |

The users rated scores of how intuitive (left), distracting (middle), and helpful (right) they found GeckoGraph. Overall, programmers consider GeckoGraph to be intuitive and not distracting. However, opinions are split on its helpfulness.

### Threats to validity

##### Task design

In this human study, most of the provided functions are very abstract. These functions are created by the authors solely for the gamified study, with a strong consideration of being puzzling and fun. They use more type variables than a typical Haskell function and are given non-descriptive names. These functions may not be representative of real-world Haskel programming.

##### The use of skips

Although we justified the use of skips in Section [5.3.5](#subsection:task), the availability of skipping does allow users to adopt more utilitarian strategies, often involving skipping a level without giving it a fair try. This happened more often in the later levels when users realized they had enough skip opportunities left to “complete the game". These strategies may result in the recorded success rates being lower than if no skips were allowed.

## Discussion

Based on the results of our user study, we analyze the strengths and weaknesses of GeckoGraph, highlighting both the positive and negative aspects of our findings. We offer insights into potential applications for type visualization tools such as GeckoGraph, suggesting future directions for their development and use.

### Strengths

The results of our experiment show that using GeckoGraph significantly affects the success rate of our participants, especially less experienced programmers. We also found no significant time difference between programming with and without GeckoGraph. To extrapolate the observed expressiveness, we speculate on the practical benefits of programming with GeckoGraph.

#### Identify the Most Important Features

One trend that we saw from the qualitative feedback is that programmers find GeckoGraph helpful for finding patterns and important features of the types. Programmers are very positive about GeckoGraph’s ability to reveal the most helpful features of a type in distinctive visual elements such as color, length, and height.

**The colors of GeckoGraph** help programmers to see the permutation of type variables in the input and output of a function. A recent review of 59 graphical perception articles showed that combining solid color hue in a filled shape provides stronger visual perception for nominal data such as type identifiers. One example of GeckoGraph’s effective use of color is the “rotation" function in the user study (Fig. [5.10](#fig:rotate)). With text-based type notation, programmers often rely on mnemonic devices such as alphabetic ordering or naming conventions. For example, the rotation function f2 :: Zero a b c d -> Zero b c d a in the game is less recognizable if changed to f2 :: Zero e v m h -> Zero v m h e. To quote a participant, “GeckoGraph is quite intuitive to see the permutations of the arguments. Also, to see how to produce and consume arguments."

|  |
| --- |
|  |

The ‘rotate’ function in level 8 of the user study. The name given in the game is ‘f2’. It shuffles the type arguments of a Zero type

**The horizontal axis of GeckoGraph** often becomes intuitive when identifying differences in function arities. For example, in Fig. [5.11](#fig:add3), the programmer intended to implement a function that sums 3 integers. In the implementation, the programmer missed a (+) function at the end; the resulting function type is largely different in length. It is also clear that the function needs to apply to one more binary function to satisfy the length requirement.

|  |
| --- |
|  |

An implementation of function add3 but the author missed an (+) from the correct implementation (.) ((+) .) (+). GeckoGraph highlights the difference in arity, and reveals that a binary function is needed on the right-hand side for the arity to match.

**The vertical axis of GeckoGraph** often sheds light on the most complex structure of this type. This can often be very useful when inspecting mismatching type errors where data types are nested. Common examples include when programmers forget to apply the value to “return" in a monadic block or to use liftIO to cast an IO effect. For example, in Fig. [5.12](#fig:maybe), the uses of return are excessive. It can be easily identified by examining the difference in the vertical layers of the two types. In text-based type notation, this is distinguished by different pairs of parenthesis. However, parenthesis is an overloaded syntax in type notation. In Haskell, parentheses are used to enclose tuples (a, b), specify the fixity (a -> b) -> c, or have no effect a -> (b -> c).

To quote some feedback from participants: “Types are much easier to understand by the GeckoGraph than by trying to parse all parenthesis and understand the types from the signature. " “It makes it easier to see at a glance when your output type is correct or what the difference between the current type and the target is."

|  |  |
| --- | --- |
|  | GeckoGraph reveals the difference in the “layers" of types] The function f is planned to have the type Maybe a -> a -> Maybe a. The programmer mistakingly applied the result to the return function, making the result inside a Monad instance. GeckoGraph reveals the difference in the “layers" of types. |

[

#### Low barrier to learn and understand

GeckoGraph has some key similarities to traditional text-based type notation. GeckoGraph respects the left-to-right reading order. GeckoGraph uses the familiar symbolic name as the secondary encoding. GeckoGraph simulates the prefix notation in type constructors and the infix notation in type operators. With these considerations, we ensured that programmers were able to use GeckoGraph fluently with a minimal amount of training.

One important class of feedback from the open section is that many programmers mentioned that they did not have any prior knowledge of Haskell but were able to solve the puzzles with the help of GeckoGraph. “It is similar enough to traditional types that it is intuitive." “This was how I parse the textual representation of types" was pointed out by multiple participants.

### Weaknesses

GeckoGraph is not without its limitations. These limitations might contribute to the ineffectiveness of providing more efficient task-solving and the limited effect on the success rate of more experienced programmers.

#### Space Usage

GeckoGraph uses horizontal space in proportion to the size of the type signature syntax tree, and GeckoGraph uses vertical space in proportion to the depth of the syntax tree. Compared to the traditional text-based language, GeckoGraph has the limitation of requiring vertical space. We have identified some approaches to minimize space usage while retaining most of the advantages of using GeckoGraph, such as displaying only the color blocks without the secondary encoding of identifier names.

#### Color Encoding

GeckoGraph relies highly on color hue as the main encoding. It provides a strong visual grouping for programmers to identify subtle patterns in types, such as the order and placement of substructures. However, the perception of color is different from person to person. This becomes an even bigger issue for color-blind or visually impaired programmers. Although GeckoGraph uses color-blind friendly schemes, it is only a method to avoid indistinguishable types and is not a strong guarantee of effectiveness. For this, we are exploring different encodings, such as patterns and shapes, to maximize the accessibility of GeckoGraph.

### Gamified Human Study

It is important to recognize that the human study is designed to be a series of puzzles. The tasks are meant to contain entertaining values. We practiced multiple gamification techniques: levels, story/theme, and goals/rewards. This not only allowed us to have confidence that participants are motivated to complete the tasks, it also lent us popularity in the Haskell community and led to a historically high participation rate. Gamification has been shown to improve engagement and motivation. This has been harnessed by many research projects to improve participation in human studies . We identify that studies on functional programming are often technical and intimidating; our use of gamification not only attracted historically high participation but also attracted a wide distribution of experience levels.

### Potential Applications

Drawing on our findings from the user study, particularly the qualitative feedback, we identify several potential applications for GeckoGraph in programming practice as well as in the teaching and learning of programming.

#### Programming Assistance

We envision many ways GeckoGraph can be integrated into programming tools. GeckoGraph can visualize and inspect types in tooltips and pop-ups. It can be used to discover the mismatching parts of two conflicting types in type errors. It can be used to generate type expressions and edit type expressions structurally. In our post-study survey, the potential integration of text editors and programming assistance were the most requested use cases proposed by the participants.

#### Documentation Assistance

From what we have learned from our human study, GeckoGraph is well suited to support the documentation of the programming library and the API documentation. It works in tandem with the traditional text-based language and can be generated mechanically, making it possible to standardize with minimal effort. For documentation sites that allow searching by name (e.g., Hoogle ), programmers often need to sieve through a list of identically named functions. For example, a simple Hoogle search for the name make shows a list of functions with vastly different usage and purpose. GeckoGraph can help speed up the selection process by providing a visual notation for each type, and programmers can use a visual grouping of colors, sizes, and positions to home in on the correct documentation page.

#### Pedagogical Applications

We believe that GeckoGraph can be a valuable tool in teaching techniques and theories in programming languages that are difficult to convey in plain language. In fact, many participants in our study reported that they had no prior knowledge of Haskell programming and that they could understand the programming concepts in the game and complete all the puzzles with the help of GeckoGraph.

Furthermore, the advanced features of GeckoGraph (Section [5.2.2](#sec:benefits)) are also suitable for teaching and learning high-level functional programming concepts. Consider the assoc function for day convolution in the Kan extension (Fig. [5.13](#fig:assoc)). Although the type signature is short, it is very difficult to trace the semantics mentally due to the number of variables, and their kinds are not obvious from the text-based notation. GeckoGraph makes understanding the type easier by visualizing the “hidden" higher-kinded types, revealing all the partially applied data types in play.

|  |
| --- |
|  |

The assoc function for day convolution in the Kan extension. Even for people who are not familiar with the exact definitions, it is easy to see that the variables f, g, and h are all high-kinded types.

## Related work

The concept of using visualization in programming languages is not new, but there has been limited research on type-level visualization specifically. In this section, we explore the topic of type visualization and place our research within the broader context of programming visualization. Finally, we examine different approaches to integrating visualization into programming tasks, including visualization as an augmentation to text-based representation and visualization as a replacement for it.

### Visualizing polymorphic types

A similar half-enclosing structure was proposed in the visualization of types by Jung . In Jung’s notation, the type constructor half encloses its arguments, but the figure for the type constructor is placed on the bottom right (Fig. [5.14](#fig:jung)). In contrast, GeckoGraph follows the natural reading order, allowing larger structures in a type signature to take precedence over smaller ones.

Compared to functions in Jung’s notation, GeckoGraph shows two major benefits. First, GeckoGraph naturally translates the general notion of a curried function. Partially, the application of a function can be read as removing the first one of its arguments. This is not the case with Jung’s notation. Second, the shape of a function type remains consistent with the shape of normal data types. In Jung’s design, a function a -> b looks sufficiently distinct from a data type f a b. This is important because, in functional languages, it is very common for abstraction to be drawn from function and normal data types. For example, functions and lists both have a functor instance, and their inner values can be altered using a fmap function. The consistent shape of GeckoGraph makes this generalization easier to see visually.

|  |
| --- |
|  |

Comparing the map function ((a -> b) -> [a] -> [b]) in text notation, GeckoGraph, and Jung’s notation.

### Visualization in programming

Using visualization techniques to improve the comprehension of polymorphic types is not new. This is often practiced to represent document properties, runtime information, and static analysis results. FluidEdt displays heap graphs in the left margin. I3 offers search similarity and change history in compact block-based diagrams. RustViz  introduced a novel visualization to aid in understanding ownership and borrowing in the Rust language. While the field of graphical type representation is relatively narrow, it has been studied to some extent. Clerici, Zoltan, and Prestigiacomo  proposed a graph-based type inference system that shows the visual representations of unification states. GeckoGraph positions itself similarly to these projects, using color, shapes, symbols, and icons to provide easy-to-glance information. However, GeckoGraph focuses on type-level information, which no other research projects do. In addition, GeckoGraph is evaluated in a much larger study than the other projects, and GeckoGraph is evaluated with a wider range of experience levels.

### Visual vs Textual representation

Many studies have compared the effectiveness of a visual-based programming environment with a textual-based one. Studies found that compared to a purely textual programming language with similar positioning, students who were taught a visual programming language show greater confidence, better retention, and enjoyment in programming courses. While showing a similar trend, GeckoGraph experiments in the context of accompanying text-based notation rather than replacing it.

Many have studied the effect of visual augmentation, providing a visual representation of programming objects without removing the text-based notation. Greenfoot allows visual and textual representations of programming concepts to be accessible to the learner. PILeT , provides a programming environment that is an adaptive presentation based on the user’s preference. Both tools show positive results in the use of visual augmentation. Although similar to GeckoGraph in combining visual language and text-based programming environment, both studies evaluated the effect based on imperative languages (Java and Python), while our evaluation focused on the effect on a functional language (Haskell).

## Conclusion and Future Work

In this paper, we propose GeckoGraph, a graphical notation for type annotations in functional programming languages. GeckoGraph aims to accompany traditional text-based type notation and to make reading, understanding, and comparing types easier. We conducted a large-scale human study using GeckoGraph compared to text-based type notation, the largest user study on functional programming we are aware of. The results of the study show that GeckoGraph helps improve programmers’ ability to succeed in programming tasks we designed, especially for novice programmers.

Our work in this area opens many new directions for future research. In particular:

* **In-the-wild Studies** Although our experiment scenarios are drawn from real-world programming tasks, a certain level of variable control is still applied to remove the effect of familiarity with the tools and libraries. However, it is necessary to assess the usefulness of tools such as GeckoGraph in terms of their real-life usage. To study this, different research methods should be used to study realistic usage and human experience. This may include field deployments or case studies.
* **Alternative Type Visualization** We strongly believe that visualization is an underutilized technique in this effort. GeckoGraph focuses on a faithful view of the tree structure of programming types. However, many more areas and types demand a human-focused approach. For instance, visualizing the ordinal relationship of subsumption or visualizing the numeric changes in dependently typed “type programs".

# Conclusion

At the beginning of this thesis, we gave an overview of the landscape of programming languages, with a special interest in functional programming and static typing. We emphasized the critical role of type errors, noting how problematic type errors can hinder both the learning and the effective use of statically typed languages. Then, we explored various Haskell type-checking methods and their evolution, setting the stage for an in-depth discussion of our interventions, ChameleonIDE, Goanna, and GeckoGraph, which are designed to deliver accurate and user-friendly error messages.

In this chapter, we review our contributions and discuss the future work that builds upon these initiatives. The planned future work encompasses what we consider a natural next step in tool design, integration with large language models (LLMs), and language availability. We provide detailed visions for how debugging type errors can be further improved using our acquired knowledge. We conclude our discussion by reaffirming our commitment to improving type errors, a core objective of this thesis, and reiterating the importance of this work within the broader field of programming language research.

## Contributions

In this thesis, we present four key areas of contribution: a categorization of type errors, ChameleonIDE, Goanna, and GeckoGraph.

### A categorization of type errors

We have developed a categorization framework for type errors based on insights into how human programmers perceive them and theories from constraint satisfiability research. This framework helps identify three critical attributes of type errors, enhancing our understanding of strategic interventions to assist programmers in effectively resolving these errors.

* **Multi-step type errors** These errors involve a sequence of logical deductions, requiring programmers to look for multiple locations of the source code to forge a coherent reasoning chain. It is crucial to clearly present these locations and their interconnected relationships within the source code to streamline this reasoning process.
* **Multi-witness type errors** These errors present an imbalance in the evidence, leading to two possible causes. Highlighting this discrepancy can guide programmers toward a more informed evaluation of the likely root causes, aiding in quicker resolution.
* **Multi-party type errors** These errors involve conflicts that present more than two potential possible types. They often indicate the coexistence of multiple underlying errors. Providing tools that can break down these errors into multiple type errors of simpler form allows programmers to tackle each error sequentially, leading to a simpler debugging workflow.

Following this classification, we delved into the three main systems we developed in our research:ChameleonIDE (see Chapter [3](#chap:chameleon)), Goanna (see Chapter [4](#chap:goanna)), and GeckoGraph (see Chapter [5](#chap:gecko-graph)). Each system is designed to tackle some different challenges associated with debugging type errors.

### Explain Multi-step type errors and the chain of thought visualization

We introduce **ChameleonIDE**, an interactive Haskell type error debugging tool. ChameleonIDE uses Minimal Unsatisfiable Subsets (MUS) as its core type error representation. Our work on ChameleonIDE is based on the existing work on Chameleon . The original Chameleon was developed as a command-line tool; its innovative approach of interactive debugging allows programmers to view different possible type error explanations and type assignments by issuing console commands. We took the idea of an MUS-based Haskell debugging tool and interactive debugging techniques and provided a modern implementation with many advanced features. Notably, ChameleonIDE enables programmers to explore the chain of reasoning behind type errors in step-by-step order. Additionally, ChameleonIDE features an adaptive user interface, allowing programmers to tailor the information density of type errors according to their experience level.

We conducted three user studies to investigate the impacts of using debugging tools that support type error slicing and interactive type error exploration, some key features provided in ChameleonIDE. Our findings reveal a notable improvement in debugging speed when employing the type error-slicing technique, particularly for complex tasks. Additionally, across the experience level, we observed a significant enhancement in debugging speed when programmers engaged with the interactive debugging features, suggesting that active exploration of type errors can positively impact the debugging process.

### Resolve multi-witness and multi-party errors with Minimal Correction Subsets

We contribute **Goanna**, a Haskell type error debugging tool that, like ChameleonIDE, utilizes type error slicing to achieve comprehensive and user-friendly error localization.

Goanna sets itself apart by employing Minimal Correction Sets (MCS) to identify potential causes of type errors. It conducts an exhaustive analysis of all potential causes and corresponding actions required to resolve the type error. In addition, Goanna employs a set of heuristics to avoid an overwhelming list of suggestions. These heuristics filter out less useful suggestions and prioritize potential causes based on their likelihood, ensuring a more focused and effective debugging experience.

To assess the efficacy of MCS-based type error debugging strategies, we compiled a dataset of 86 Haskell programs sourced from various online discussions, embodying a wide range of type error scenarios. We evaluated the accuracy of Goanna in identifying and resolving type errors in comparison to traditional compiler tools. Our findings affirm that Goanna consistently provides more accurate error-cause identification compared to other tools. Further, our heuristics not only effectively narrow down the list of possible causes but also consistently include the real cause in its top recommendations. Lastly, although Goanna is slower than the conventional tools, it remains well within a range suitable for providing real-time feedback.

### Visualizing Types

We introduce **GeckoGraph**, an innovative graphic notation system designed specifically for Haskell types. Unlike traditional type signatures, GeckoGraph utilizes a combination of colors, shapes, and symbols to highlight distinct type structures. GeckoGraph provides a clear visual representation for type-level features such as type classes, parametric type variables, and high-rank types. When programmers need to compare two types, a frequent requirement in resolving type errors, GeckoGraph provides strong visual grouping that is often able to underscore subtle differences, providing clear graphical distinctions.

We conducted a large-scale user study to evaluate the effectiveness of GeckoGraph in enhancing the traditional text-based approach to type signatures. The study was designed as an interactive puzzle game incorporating gamification elements to stimulate participation and engagement. In total, 721 programmers of all experience levels participated in the study. While the results showed that the use of GeckoGraph did not significantly impact the speed or overall success rate of solving problems, it proved significant in supporting beginners in solving harder tasks more successfully. Additionally, feedback collected through a qualitative post-study survey was positive, suggesting that GeckoGraph is intuitive, non-intrusive, and helpful.

## Future Work

In this section, we present three promising directions stemming from our existing work. Firstly, drawing upon our prior experience of researching type debugging tools, we now sketch a novel debugging tool: *TypeTutor*. We envision the interfaces and interactions within *TypeTutor*, which have been shaped by insights from our existing tools and feedback from various user studies.

Secondly, we delve into the realm of rapid advancements in large language models (LLMs) and their evolving role in programming assistance, distinguishing them from conventional theory-based tools like ChameleonIDE and Goanna. We explore the possibility of integrating traditional tools with LLMs to effectively capitalize on their unique strengths and address their weaknesses.

Lastly, we outline our plans to adapt our debugging tools for use in other programming languages, underscoring the potential advantages and challenges associated with this endeavor.

### TypeTutor: Question-Based Type Debugging

By observing programmers tackle type error challenges in a series of user studies, we gained valuable insights into Haskell programmers’ debugging habits and approaches toward challenging type errors. Armed with this understanding, we are inspired to envision a novel debugging tool named *TypeTutor*. The core concept of *TypeTutor* would be to map debugging tasks into a series of questions that programmers would naturally ask while also to help programmers break down high-level debugging questions into actionable, granular queries.

This style of programming is often referred to as “natural programming environment” . Previous studies have underscored the benefits of employing a natural debugging interface, exemplified by tools such as Alice  and WhyLine . However, existing research predominantly concentrates on debugging runtime errors, neglecting exploration into type-level debugging. Leveraging our experience with ChameleonIDE, Goanna, and GeckoGraph, we envision meaningful progress in this domain.

#### Why questions

A fundamental aspect of debugging type errors involves asking questions such as "Why does the type error occur?" and "Why is the expected type X?" These "why" questions highlight gaps in understanding where a thorough explanation is in demand. *TypeTutor* will intelligently address this by displaying all relevant "why" questions when a programmer hovers over an expression of interest. Upon selecting a question, *TypeTutor* will promptly provide the corresponding detailed answer (Fig. [6.1](#fig:why)).

|  |
| --- |
|  |

The figure shows how *TypeTutor* would provide programmers valid ‘why’ questions to ask in a type error. When hovering on a location that is marked as a type error, *TypeTutor* will identify an ‘expected’ type (Top) and an ‘actual’ type (Bottom). *TypeTutor* would also prompt programmers to interrogate each branch to understand how these conclusions are drawn by hovering on the ‘Why’ buttons.

#### Follow-up questions

In the context of debugging type errors, particularly multi-step type errors, it proves beneficial to enable programmers to incrementally uncover the underlying inference logic. *TypeTutor* will support this process by allowing programmers to ask follow-up questions. Typically, these questions build upon the responses to earlier inquiries. Follow-up questions in *TypeTutor* would facilitate tasks akin to the interactive debugging steps in ChameleonIDE. However, unlike ChameleonIDE, *TypeTutor* would not need a separate interface for such a task. Instead, when a response includes the option for further inquiry, *TypeTutor* would provide a follow-up hint at the end of the answer, encouraging programmers to continue tracing the root cause (Fig. [6.2](#fig:follow-up)). This integrated approach would help maintain a streamlined user experience while enabling deep exploration of type errors.

|  |
| --- |
|  |

Following the case in Fig. [6.1](#fig:why), *TypeTutor* will prompt programmers to ask follow-up questions. In this case, *TypeTutor* will inform the programmer that matchFirst on line 3 has the type Int -> (Int, Int) -> Bool can be inferred from two pieces of clues: the definition of matchFirst and the filter on line 3 instantiated as ((Int, Int) -> Bool) -> [(Int, Int)] -> [(Int, Int)]. Programmers can trace further by following the hints again in the answer section.

#### How questions

’How’ questions in debugging focus on providing prescriptive guidance regarding program errors. Such questions do not solely depend on logical precision; they require understanding the knowledge gaps and delivering clear, followable instructions. *TypeTutor* would aid programmers by facilitating questions on how to rectify specific type errors. In Goanna, the Maximal Satisfiable Subsets (MSS) analysis can suggest the recommended type for each possible correction. *TypeTutor* would advance this concept by offering examples of syntax changes tailored to these recommended types. In the example (Fig. [6.3](#fig:how)), *TypeTutor* would provide various examples of how to change ’1’ to an Integer type, including changing it to an integer literal, an integer variable, or an expression that evaluates to an integer.

|  |
| --- |
|  |

Following the case in Fig. [6.1](#fig:why), *TypeTutor* will provide instructions on how to fix the type error. In detail, *TypeTutor* will provide various examples of how to change ’1’ to an Integer type, including changing it to an integer literal, an integer variable, or an expression that evaluates to an integer.

#### What-if questions

Finally, as we have identified, type errors frequently involve multiple potential causes and explanations. When integrated into a question-and-answer interface, the presence of multiple causes naturally prompts counterfactual questions such as "What are other ways that can cause this type error?" Recognizing this, *TypeTutor* will provide a natural and user-friendly interface for exploring various potential causes and solutions. This approach, while similar to the functionality offered by Goanna, features a more concise and intuitive interface for programmers to switch between different potential causes. With *TypeTutor*, programmers will find the option: ‘Other ways to fix the error’ in every type error question list; hovering over the option will present alternative explanations and resolution plans (Fig. [6.4](#fig:what-if)).

|  |
| --- |
|  |

If the programmer disagrees that the literal ’1’ is the cause of the error and should be changed, *TypeTutor* will provide alternative explanations and resolution plans. When hovering on the option ‘Other ways to fix the error’, *TypeTutor* will show three other potential fixes (Top). Choosing any of them, say the first option, will change the type error mark to another set of locations, and different explanations are provided when asking the ‘why’ question at the new type error.

### Integration with Large Language Models

At the time of writing this thesis, LLMs were actively experimented with performing all kinds of tasks that require human creativity, including programming. Many attempts have been made to use LLMs in programming tasks , intending to free programmers for higher-level thinking. This development is tangential to our objective of explaining type errors and reasoning about the logic of type inference.

|  |
| --- |
|  |

The figure shows the user prompted 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 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 multi-step reasoning ability. Fig. [6.5](#fig:llm) 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.

|  |
| --- |
|  |

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. [6.6](#fig:llm2)). 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 , 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 . We believe in the great potential of integrating LLMs with existing theory-based tools, such as ChameleonIDE and Goanna. This integration could enhance the performance of LLMs by aligning their operations with accurate theoretical guides or utilizing their strengths in areas where traditional tools fall short, such as suggesting syntactical improvements. This synergy could lead to more robust programming assistance using both types of tools. We propose two candidate workflows:

### Support for Other Languages

Haskell serves as a vital platform for the experiments of programming languages and type systems. We firmly believe that the next critical phase in our research involves expanding our tools to encompass other languages. Recently, we observed that the challenges of overcoming complex type errors have permeated several mainstream languages, including Rust and TypeScript . By integrating our debugging tools into these languages, we would be able to engage a broader audience and subject our tools to a diverse range of software projects varying in style and scale. This will not only increase the applicability of our tools but also provide us with extensive insights into how to further improve our design.

Much of our work has already been conceived with the adaptability to multiple languages in mind. For example, GeckoGraph is designed to be language-agnostic, allowing for implementations in different programming languages. However, adapting tools like ChameleonIDE and Goanna presents certain complexities. Although the analysis of unsatisfiable constraints and the theories and implementations concerning the enumeration of Minimal Unsatisfiable Subsets (MUSes) and Minimal Correction Subsets (MCSes) are universally applicable across all languages, specific challenges arise when taking into account the nuances of individual languages.

The first challenge involves constraint generation. Each language requires the development of specific rules for generating constraints and conforming to its own typing rules, which could necessitate substantial modifications depending on the type-level features each language possesses. For instance, all functions in Haskell are curried, meaning that they can be provided with fewer arguments than they are defined, a characteristic absent in TypeScript, where functions are defined with fixed arity and can be overloaded. This discrepancy means that in TypeScript, a type error can occur if a function is applied to fewer arguments than it is defined for, an error that would not typically arise in Haskell.

The second challenge concerns the presentation of type errors, which may need nuanced adjustments for different languages. For example, novel visualization techniques may be required to effectively convey concepts absent in Haskell, such as row polymorphism in TypeScript or lifetime constraints in Rust. By addressing these challenges, we can truly leverage the potential of our tools across various programming environments.

#### Supporting Dependently Typed Languages

As mentioned in previous chapters, dependently typed languages like Idris and Agda represent a rigorous subclass within the realm of statically typed languages. These languages uniquely enable the type-level computation of values and types, allowing programmers to embed fine-grained and precise constraints to ensure program correctness at compile time. Due to their ability to directly encode program behavior into types, programs written in dependently typed languages can often be validated purely through the type-checking process. This makes them ideal candidates for code generation through LLMs, where programmers can perform checks on whether they conform to the provided specifications.

However, despite their potential, dependently typed languages currently hold a modest position in the mainstream programming landscape. We believe that enhancing our type debugging tools to support dependently typed languages can bring a step forward in improving their learnability and adoption, especially among programmers who are new to dependent typing. As type-level computations grow in complexity, our tools, designed to aid in understanding and reasoning, become increasingly valuable.

## Conclusion

The field of programming languages is truly captivating, representing a confluence of ideas from various disciplines and schools of thought. Among the myriad concepts, functional programming and static typing are particularly important for their profound impact on the domain. My research aims to highlight the benefits of functional programming and static typing while also addressing the challenges they pose. By refining how types and type errors are presented and explained, we strive to make functional programming and static typing accessible and user-friendly to programmers.

Our methodologies are largely grounded in theories of constraint satisfiability. These include analyzing Minimal Unsatisfiable Subsets (MUS) through tools like ChameleonIDE and Minimal Correction Subsets (MCS) via Goanna. In addition, we employ various human-centered research techniques, such as formative studies, user studies, and rapid prototyping. These approaches enhance the practicality of our research and, hopefully, its relevance to real-world programming.

We are convinced that type error enhancement and explanation is a valuable trajectory for research in programming languages. It is our hope that our work will serve as a helpful reference, inspiring future studies that continue to augment and expand our arsenal of type error debugging tools.

# Game Levels In User Study ZeroToHero

We provide all the level settings we used in our user study. The online game is still open source and available for evaluation . However, this can be attempted locally with a Haskell interpreter or even with a pen and paper. The target type is the desired type signature for the function zeroToHero. The available functions show a list of functions that are allowed to be used in the implementation. It is not required to use all the available functions, and it is not forbidden to use any other functions or variables outside the provided functions; even the Haskell prelude is not available.

## Level 1: Trial run

### Target Type

* zeroToHero :: Zero a -> Hero a

### Available Functions

* f :: Zero a -> Hero a

### Possible Solution

* zeroToHero z = f z

## Level 2: Assembly required

### Target Type

* zeroToHero :: Zero a -> Hero a

### Available Functions

* runZero :: Zero a -> a
* mkHero :: a -> Hero a
* ($) :: (a -> b) -> a -> b

### Possible Solution

* zeroToHero z = mkHero (runZero z)

## Level 3: Which path?

### Target Type

* zeroToHero :: Zero a -> Hero (a, a)

### Available Functions

* f1 :: Zero a -> Hero a
* f2 :: Zero a -> (a, a)
* f3 :: Hero a -> Hero (a, a)
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = f3 . f1 $ z

## Level 4: A repeating pattern

### Target Type

* zeroToHero :: Zero a b -> Hero b b

### Available Functions

* f1 :: Zero a b -> Hero b a
* f2 :: Zero a a -> Hero a a
* f3 :: Zero a b -> Zero b a
* f4 :: Zero a b -> Zero b b
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = f2 . f4 $ z

## Level 5: A perfect pair

### Target Type

* zeroToHero :: Zero a b -> Hero b b

### Available Functions

* fst :: (a, b) -> a
* snd :: (a, b) -> b
* f1 :: Zero a b -> Hero b a
* f2 :: Zero a a -> Hero a a
* f3 :: Zero a b -> Zero b a
* f4 :: Zero a b -> Zero b b
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = snd .f3 . f1 $ z

## Level 6: Monty Hall

### Target Type

* zeroToHero :: Zero a b c -> Hero c a

### Available Functions

* f1 :: Zero a b c-> Zero c b a
* f2 :: Zero a b c -> Zero a c c
* f3 :: Zero a b c -> Hero b c
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = f3 . f1 . f2 $ z

## Level 7: TIE fighter

### Target Type

* zeroToHero :: Zero a b c -> Hero c

### Available Functions

* f1 :: Zero a b c -> Hero (a -> b)
* f2 :: Zero a b c -> Hero (b -> c)
* f3 :: Zero a b c -> Hero a
* (<$>) :: (a -> b) -> Hero a -> Hero b
* (<\*>) :: Hero (a -> c) -> Hero a -> Hero c
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = f2 z <\*> (f1 z <\*> f3 z)

## Level 8: The middle man

### Target Type

* zeroToHero :: (a -> d) -> (b -> d) -> (c -> d) -> Zero a b c -> Hero a d c

### Available Functions

* f1 :: Zero a b c -> Zero c a b
* f2 :: Zero a b c -> Hero a b c
* fmap :: (c -> d) -> Zero a b c -> Zero a b d
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero ad bd cd z = f2 . f1 . f1 . fmap bd . f1 $ z

## Level 9: Split the difference

### Target Type

* zeroToHero :: Zero a b c d -> Hero d d d d

### Available Functions

* f1 :: Zero a b c -> Zero c a b
* f2 :: Zero a b c -> Hero a b c
* fmap :: (c -> d) -> Zero a b c -> Zero a b d
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero ad bd cd z = f2 $ f1 $ f1 $ f3 $ z

## Level 10: The roller coaster

### Target Type

* zeroToHero :: Zero (a -> b -> c -> d) a b c -> Hero d

### Available Functions

* f1 :: Zero (a -> b) a c d -> Zero () b c d
* f2 :: Zero a b c d -> Zero b c d a
* f3 :: Zero a b c d -> Hero d
* ($) :: (a -> b) -> a -> b
* (.) :: (b -> c) -> (a -> b) -> a -> c

### Possible Solution

* zeroToHero z = f3 . f2 . f2 . f1 . f2 . f1 . f2 . f1 $ z