ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration

Shuai Fu
Faculty of Information Technology
Monash University
Clayton, Australia
shuai.fu@monash.edu

Tim Dwyer
Faculty of Information Technology
Monash University
Clayton, Australia
tim.dwyer@monash.edu

Peter J. Stuckey
Faculty of Information Technology
Monash University
Clayton, Australia
peter.stuckey@monash.edu

Jackson Wain
Faculty of Information Technology
Monash University
Clayton, Australia
ORCID: 0000-0003-2006-3538

Jesse Linossier
Faculty of Information Technology
Monash University
Clayton, Australia
ORCID: 0000-0001-6782-7019

Abstract—**Dynamically typed programming languages are popular in education and the software industry. While presenting a low barrier to entry, they suffer from runtime type errors and longer-term problems in code quality and maintainability. Statically typed languages, while showing strength in these aspects, lack in learnability and ease of use. In particular, fixing type errors poses challenges to both novice users and experts. Further, compiler type error messages are presented in a static way that is biased toward the first occurrence of the error in the program code. To help users resolve such type errors we introduce ChameleonIDE, a type debugging tool that presents type errors to the user in an unbiased way, allowing them to explore the full context of where the errors could occur. Programmers can interactively verify the steps of reasoning against their intention. Through three studies involving actual programmers, we showed that ChameleonIDE is more effective in fixing type errors than traditional text-based error messages. This difference is more significant in harder tasks. Further, programmers actively using ChameleonIDE's interactive features are shown to be more efficient in fixing type errors than passively reading the type error output.**

Index Terms—**types, type errors, debugging, visualization, exploration**

Dynamically typed programming languages such as JavaScript and Python have risen in popularity in recent decades [1]. These languages present a low barrier of entry, especially to beginner programmers: they require no type declaration, variable types or object structures can be modified dynamically, and functions can deal with dynamic input using ad-hoc polymorphism and runtime reflection. However, studies show that dynamically typed languages negatively affect development productivity [2], code usability [3], and code quality [4]–[6]. They are often found to produce error-prone code [7]–[9] and require strong programmer discipline to avoid pitfalls [7]. For these reasons, many modern dynamically-typed languages have introduced static typing annotations as part of the core language features in recent years (e.g. *TypeScript* [10] and *mypy* [11]).

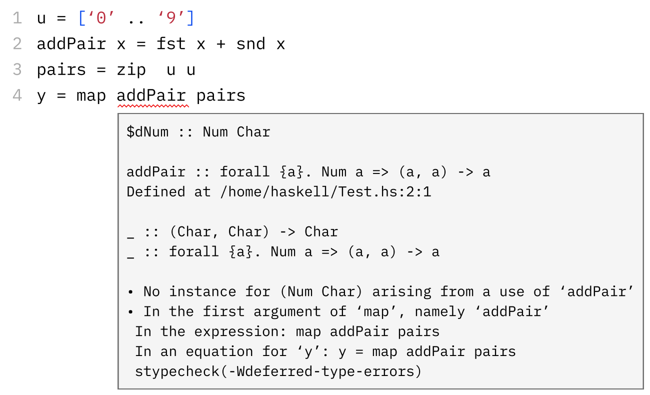
Functional programming languages have long enjoyed rigorous type systems and expressive type-level features. Techniques such as type inference and algebraic types have been standard practice for decades in functional languages such as ML and Haskell, and more recently in multi-paradigm languages, such 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" [12]. Type classes, an implementation of generic programming, were introduced to Haskell in 1988 [12], and now can be found in most popular languages such as C# [13], Java [14], and TypeScript [15].

One crucial challenge of programming in statically-typed languages is that type errors can sometimes be difficult to resolve [16], [17]. 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 [18].

This paper 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 II-A0a), the candidate expression card (Section II-A0b), and the deduction step (Section II-A0c). 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 [19].

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.



```
1 u = ['0' .. '9']
2 addPair x = fst x + snd x
3 pairs = zip  u u
4 y = map addPair pairs
```

```
$dNum :: Num Char

addPair :: forall {a}. Num a => (a, a) -> a
Defined at /home/haskell/Test.hs:2:1

_ :: (Char, Char) -> Char
_ :: forall {a}. Num a => (a, a) -> a

• No instance for (Num Char) arising from a use of 'addPair'
• In the first argument of 'map', namely 'addPair'
 In the expression: map addPair pairs
 In an equation for 'y': y = map addPair pairs
 stypecheck(-Wdeferred-type-errors)
```

Fig. 1. A type error displayed in Visual Studio Code [21] and the Haskell Vscode extension [22]. The expression `addPair` is blamed for causing the type error. This may not match the programmers' intention.

- 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..

I. Motivation

The design requirements of ChameleonIDE are motivated by limitations of traditional type errors, as documented in a number of studies (e.g. [17], [20]), but which we illustrate here with a few motivating examples.

*1) **Traditional type errors show only limited location**:* Haack and Wells [23] 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. 1 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. In this small example, the whole context is visible, but it can become problematic in large programs where the lines contributing to the type error are far apart in the source code.

*2) **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. 1, 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.

*3) **Traditional type errors give poor explanations**:* When the compiler rejects a program, the internal state of type checking is the result of a complex computation. But 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. 1, the evidence for the type error is gathered from the previous declarations. These have to be rediscovered by programmers using less rigorous methods.

A. 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'.

II. Chameleon IDE

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 II-B.

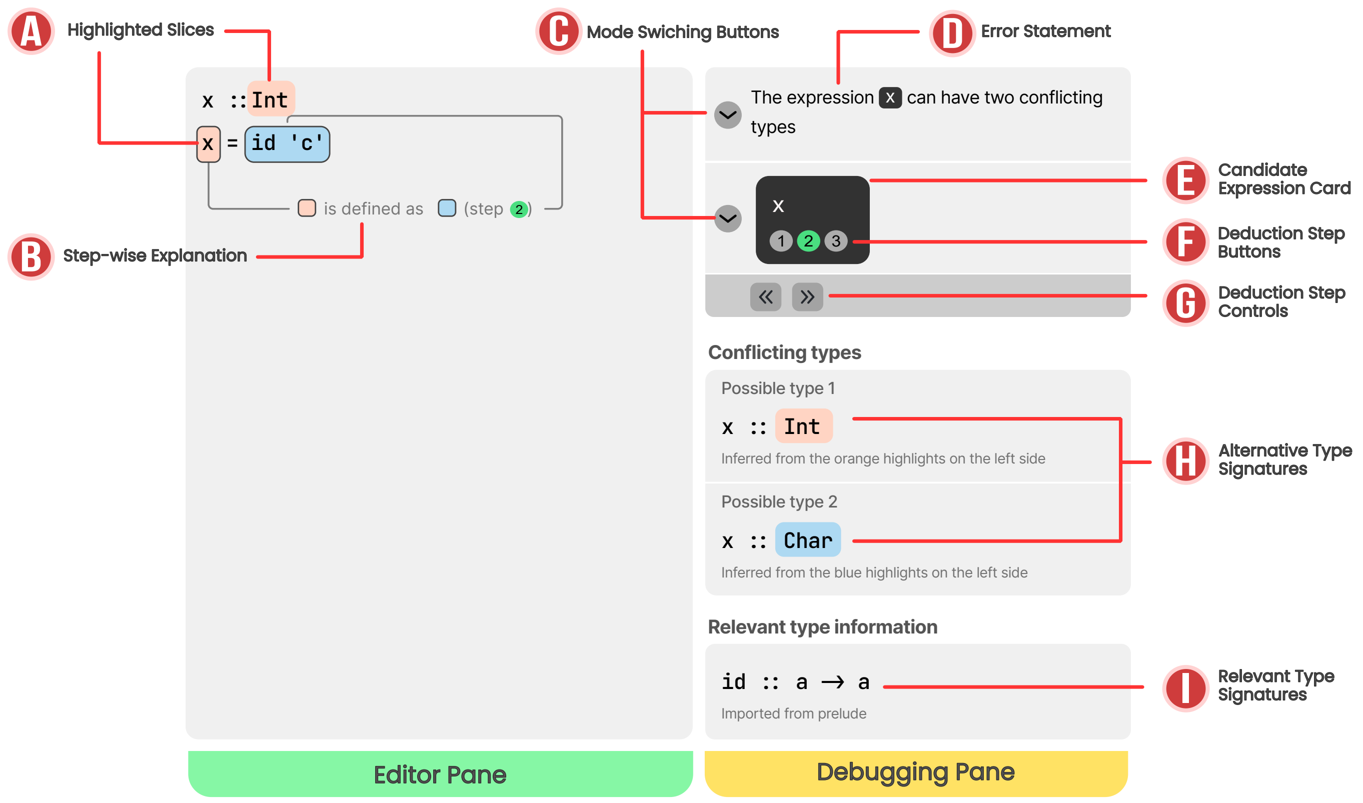
A. The Debugging Interface

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

a) 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). 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.

b) 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. 4, `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



```
x  :: Int
x = id 'c'
```

is defined as (step 2)

The expression x can have two conflicting types

Conflicting types

Possible type 1

`x :: Int`

Inferred from the orange highlights on the left side

Possible type 2

`x :: Char`

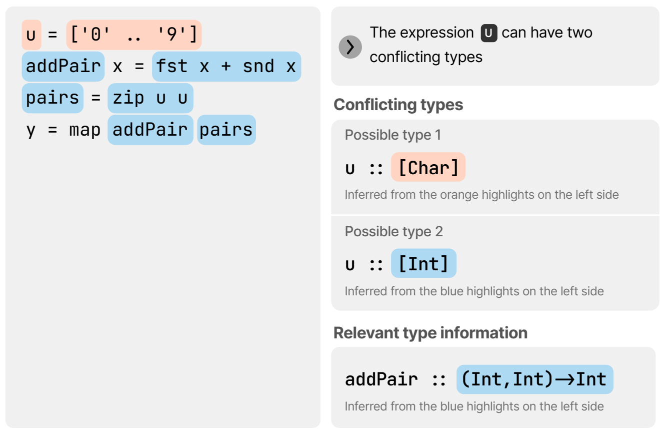
Inferred from the blue highlights on the left side

Relevant type information

`id :: a → a`

Imported from prelude

Fig. 2. **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.



```
u = ['0' .. '9']
addPair x = fst x + snd x
pairs = zip u u
y = map addPair pairs
```

The expression u can have two conflicting types

Conflicting types

Possible type 1

`u :: [Char]`

Inferred from the orange highlights on the left side

Possible type 2

`u :: [Int]`

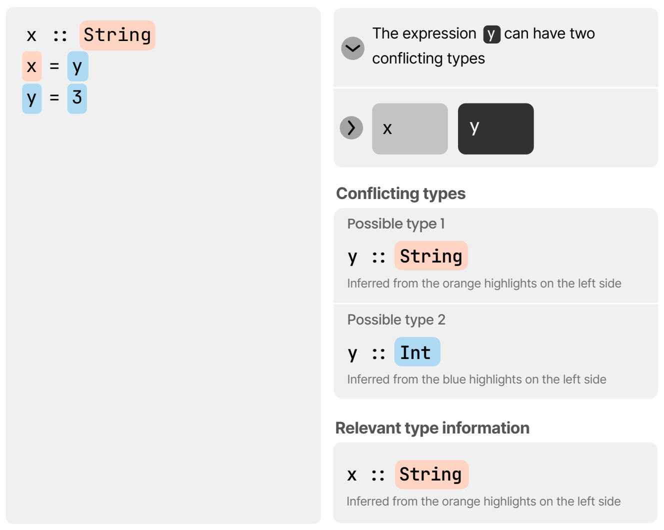
Inferred from the blue highlights on the left side

Relevant type information

`addPair :: (Int,Int)→Int`

Inferred from the blue highlights on the left side

Fig. 3. **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. 1.



```
x  :: String
x = y
y = 3
```

The expression y can have two conflicting types

Conflicting types

Possible type 1

`y :: String`

Inferred from the orange highlights on the left side

Possible type 2

`y :: Int`

Inferred from the blue highlights on the left side

Relevant type information

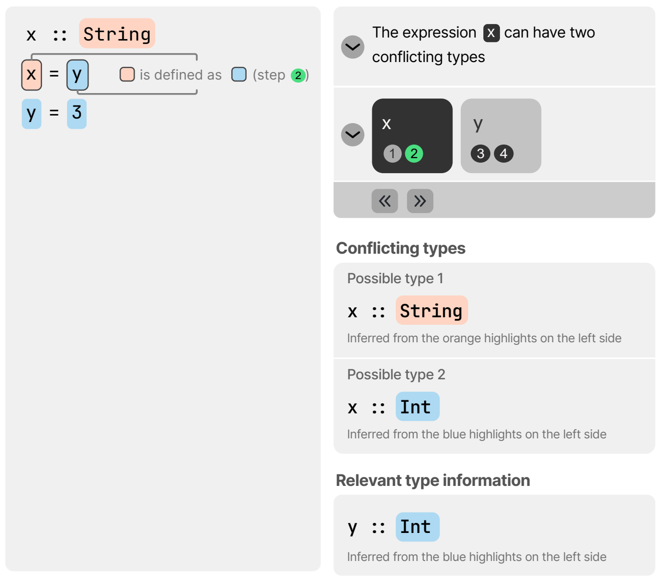
`x :: String`

Inferred from the orange highlights on the left side

Fig. 4. **ChameleonIDE with candidate expression cards enabled.** Indicates the type error can occur in the definition of x or y.

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.

c) Deduction steps: Deduction steps allow programmers to explore all the error locations one at a time (Fig. 5). Steps are shown as a list of sequentially numbered circular buttons (step buttons) and an explanation layer in the editor window. In



```
x :: String
x = y     is defined as     (step 2)
y = 3
```

The expression x can have two conflicting types

Conflicting types

Possible type 1

`x :: String`

Inferred from the orange highlights on the left side

Possible type 2

`x :: Int`

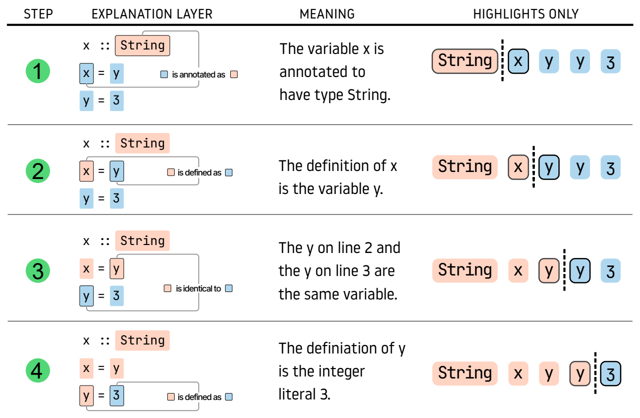
Inferred from the blue highlights on the left side

Relevant type information

`y :: Int`

Inferred from the blue highlights on the left side

Fig. 5. **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.



STEP	EXPLANATION LAYER	MEANING	HIGHLIGHTS ONLY
1	x :: String / x = y / y = 3 / is annotated as	The variable x is annotated to have type String.	String ⋮ x y y 3
2	x :: String / x = y / y = 3 / is defined as	The definition of x is the variable y.	String x ⋮ y y 3
3	x :: String / x = y / y = 3 / is identical to	The y on line 2 and the y on line 3 are the same variable.	String x y ⋮ y 3
4	x :: String / x = y / y = 3 / is defined as	The definiation of y is the integer literal 3.	String x y y ⋮ 3

Fig. 6. 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. 2-G) or by directly clicking on a step button (Fig. 2-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.

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

TABLE I
CHAMELEONIDE MODES AND FEATURES

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. 5 generates a list of steps shown in Fig. 6 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.

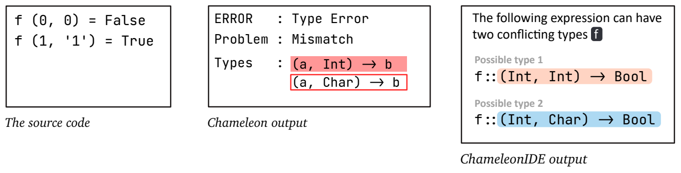
Internally, deduction steps are different ways to divide the error locations into two groups, denoted by the two colors. Each color infers a different type of the candidate expression. Each increment/decrement of the step changes the splitting point (dotted lines in Fig. 6) of the two colors.

d) 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 [24]. 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. 2-C). The features accessible from different modes are summarized in table I.

B. 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) [25], 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 et al. [26] 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



```
f (0, 0) = False
f (1, '1') = True
```

The source code

```
ERROR   : Type Error
Problem : Mismatch
Types   : (a, Int) → b
          (a, Char) → b
```

Chameleon output

The following expression can have two conflicting types f

Possible type 1

```
f::(Int, Int) → Bool
```

Possible type 2

```
f::(Int, Char) → Bool
```

ChameleonIDE output

Fig. 7. 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`.

out of date. Our ChameleonIDE implementation extends the original Chameleon approach in a number of ways.

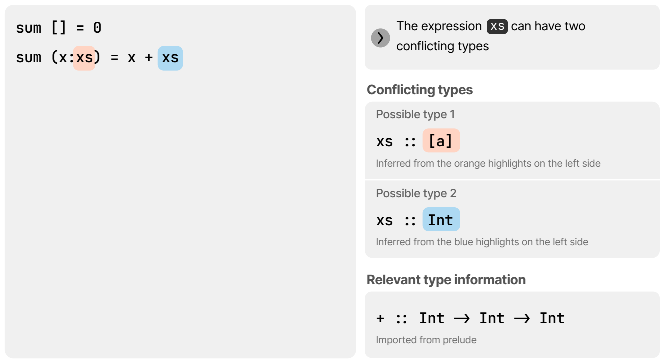
a) Recovering concrete types from type errors: Using only constraints from the MUS is sufficient to locate the type error, but to recover types from type errors we need constraints from parts of the program that are irrelevant to the type error. For instance, consider an ill-typed 2-tuple where two possible 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. 7. 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.

b) Type error explanation: In addition, ChameleonIDE provides support for type explanation. Similar to the type explanation system in [27], 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.

```
if a then b else c
a = "True"
```

Listing 1. 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 Listing 1, ChameleonIDE generates the following constraints and labels (in brackets) $T_a = Bool$ (if condition), $T_b = T_c$ (if branches), $T_a = String$ (definition). Clearly, as T_a 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 Listing 1 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 II-A0c).



```
sum [] = 0
sum (x:xs) = x + xs
```

The expression xs can have two conflicting types

Conflicting types

Possible type 1

```
xs :: [a]
```

Inferred from the orange highlights on the left side

Possible type 2

```
xs :: Int
```

Inferred from the blue highlights on the left side

Relevant type information

```
+ :: Int → Int → Int
```

Imported from prelude

Fig. 8. Maxine's code to calculate the sum of a list of integers; ChameleonIDE reports an error on the expression `xs`.

III. 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.

A. Basic mode

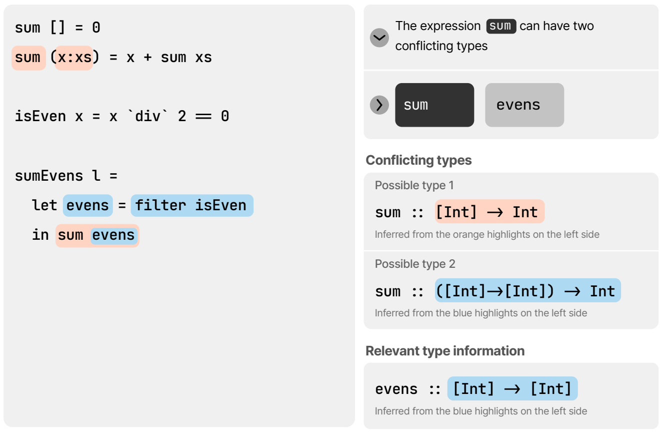
Maxine writes a function to calculate the sum of a list of numbers, but ChameleonIDE shows there is a type error (Fig. 8). After reading the error reports, Maxine realizes that the error revolves around the expression `xs`. That is: `xs` can be either `[a]` or `Int`. By matching the color in the conflicting type block (Fig. 2-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.

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

B. 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. 9). 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`.

Maxine therefore clicks on the `evens` card and ChameleonIDE reports two possible types for the expression `[Int]` and `[Int] -> [Int]` (Fig. 10). 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.



```
sum [] = 0
sum (x:xs) = x + sum xs

isEven x = x `div` 2 == 0

sumEvens l =
  let evens = filter isEven
  in sum evens
```

The expression sum can have two conflicting types

sum evens

Conflicting types

Possible type 1

`sum :: [Int] → Int`

Inferred from the orange highlights on the left side

Possible type 2

`sum :: ([Int]→[Int]) → Int`

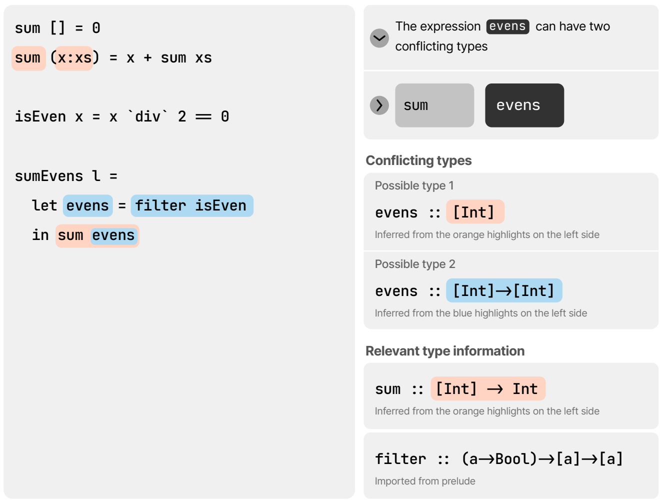
Inferred from the blue highlights on the left side

Relevant type information

`evens :: [Int] → [Int]`

Inferred from the blue highlights on the left side

Fig. 9. Maxine's code to calculate only the sum of even numbers. ChameleonIDE reports an error with two candidate expressions.



```
sum [] = 0
sum (x:xs) = x + sum xs

isEven x = x `div` 2 == 0

sumEvens l =
  let evens = filter isEven
  in sum evens
```

The expression evens can have two conflicting types

sum evens

Conflicting types

Possible type 1

`evens :: [Int]`

Inferred from the orange highlights on the left side

Possible type 2

`evens :: [Int]→[Int]`

Inferred from the blue highlights on the left side

Relevant type information

`sum :: [Int] → Int`

Inferred from the orange highlights on the left side

`filter :: (a→Bool)→[a]→[a]`

Imported from prelude

Fig. 10. 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.

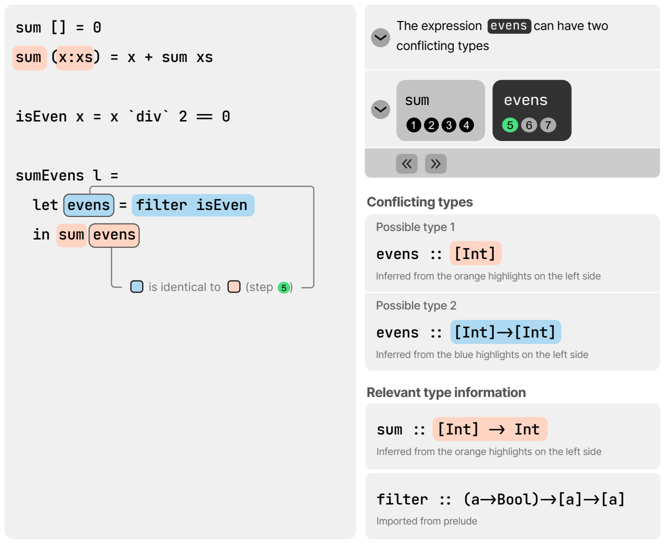
C. Advanced mode

To illustrate the deduction steps with the task shown in section III-B, first, Maxine clicks on step 5 (Fig. 11) 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. 12) and verifies that `evens` should be the same type as the declaration on the right-hand side.

Lastly, Maxine clicks on step 7 (Fig. 13), 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.

IV. Evaluation

We conducted three user studies, iteratively refining the ChameleonIDE UI and evaluating several research questions as per Fig. 14.



```
sum [] = 0
sum (x:xs) = x + sum xs

isEven x = x `div` 2 == 0

sumEvens l =
  let evens = filter isEven
  in sum evens
```

is identical to (step 5)

The expression evens can have two conflicting types

sum 1 2 3 4
evens 5 6 7

« »

Conflicting types

Possible type 1

`evens :: [Int]`

Inferred from the orange highlights on the left side

Possible type 2

`evens :: [Int]→[Int]`

Inferred from the blue highlights on the left side

Relevant type information

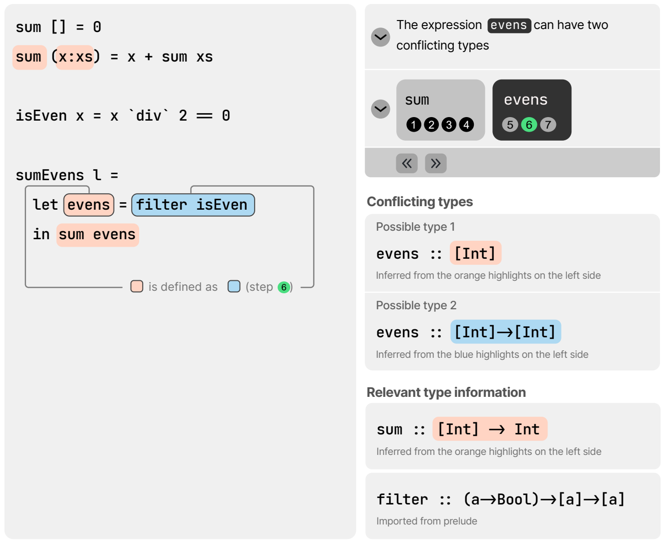
`sum :: [Int] → Int`

Inferred from the orange highlights on the left side

`filter :: (a→Bool)→[a]→[a]`

Imported from prelude

Fig. 11. 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.



```
sum [] = 0
sum (x:xs) = x + sum xs

isEven x = x `div` 2 == 0

sumEvens l =
  let evens = filter isEven
  in sum evens
```

is defined as (step 6)

The expression evens can have two conflicting types

sum 1 2 3 4
evens 5 6 7

« »

Conflicting types

Possible type 1

`evens :: [Int]`

Inferred from the orange highlights on the left side

Possible type 2

`evens :: [Int]→[Int]`

Inferred from the blue highlights on the left side

Relevant type information

`sum :: [Int] → Int`

Inferred from the orange highlights on the left side

`filter :: (a→Bool)→[a]→[a]`

Imported from prelude

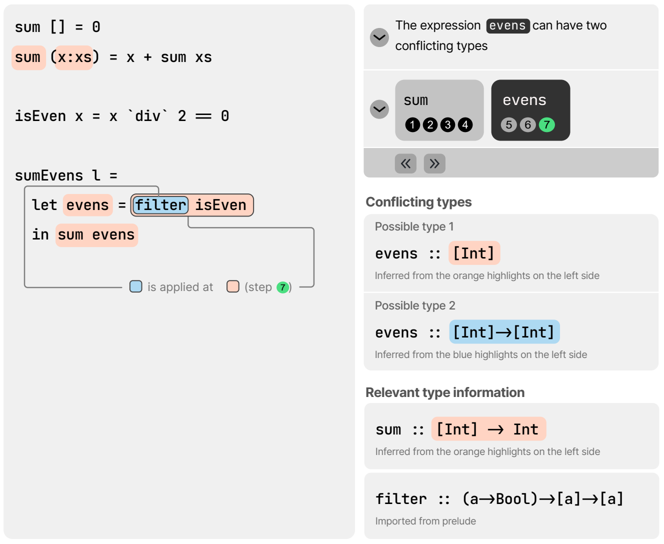
Fig. 12. In step 6, ChameleonIDE explains that `evens` is defined as the expression `filter isEven`. The left-hand side and the right-hand side should have the same type.

A. 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



```
sum [] = 0
sum (x:xs) = x + sum xs

isEven x = x `div` 2 == 0

sumEvens l =
  let evens = filter isEven
  in sum evens
```

The expression evens can have two conflicting types

Conflicting types

Possible type 1

```
evens :: [Int]
```

Inferred from the orange highlights on the left side

Possible type 2

```
evens :: [Int]→[Int]
```

Inferred from the blue highlights on the left side

Relevant type information

```
sum :: [Int] → Int
```

Inferred from the orange highlights on the left side

```
filter :: (a→Bool)→[a]→[a]
```

Imported from prelude

Fig. 13. 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.

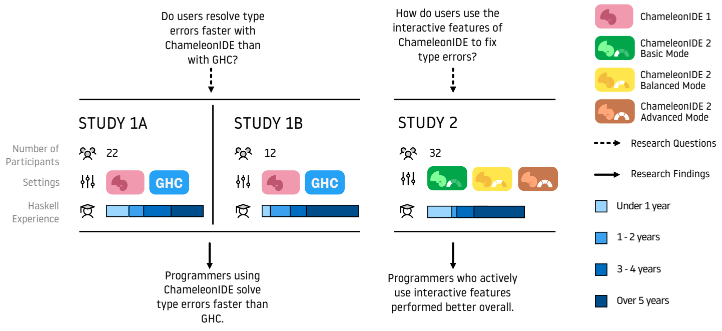

Do users resolve type errors faster with ChameleonIDE than with GHC?
How do users use the interactive features of ChameleonIDE to fix type errors?
STUDY 1A
STUDY 1B
STUDY 2
Number of Participants
Settings
Haskell Experience
22
12
32
GHC
GHC
Programmers using ChameleonIDE solve type errors faster than GHC.
Programmers who actively use interactive features performed better overall.
ChameleonIDE 1
ChameleonIDE 2 Basic Mode
ChameleonIDE 2 Balanced Mode
ChameleonIDE 2 Advanced Mode
Research Questions
Research Findings
Under 1 year
1 - 2 years
3 - 4 years
Over 5 years

Fig. 14. The timeline of ChameleonIDE evaluation.

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 difference 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 receives a single Haskell file that contains one or more type errors. They were then asked to correct the code with the help of 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 also passes all the functional tests. Participants are able to 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 [28] to record the

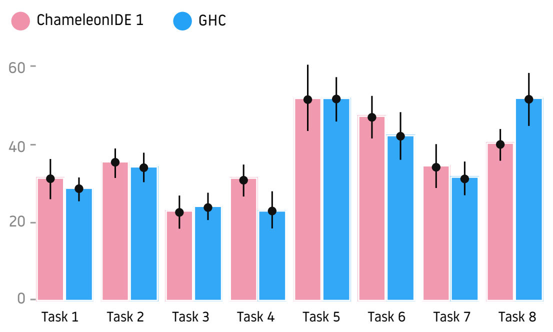


Fig. 15. Study 1a task completion time (secs.) with 95% confidence interval.

study sessions. This allows us to identify usability issues in the study and to recognize general patterns.

B. ChameleonIDE Human Studies

*1) **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 choose 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 [29]. 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 [16].

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

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

***Results**:* The data collected during study 1a, Fig. 15 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

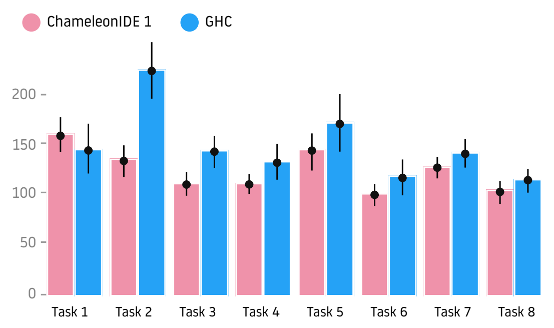


Fig. 16. Study 1b task completion time (secs.) with 95% confidence interval.

Interaction level	Description
Minimal	Users completed the tasks by making changes in source code, type checking, and reading error messages.
Low	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.
High	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.

TABLE II
LEVELS OF PROGRAMMER INTERACTION AND THEIR DESCRIPTION

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 (figure 16), 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: $t(23) = -3.86, p = 0007$. 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.

*2) **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 [30] 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 through the three different modes and repeated three cycles in nine tasks. The order of the three modes in each cycle is counterbalanced among all participants. However, participants can 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 II.

As shown in Fig. 17, 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 ($p \leq 0.001$, 95% C.I. = [18.26, 31.41]), and between the minimal interaction group and the low interaction group ($p \leq 0.001$, 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.

Another observation is when using the mode switching feature of ChameleonIDE, we show this by presenting the

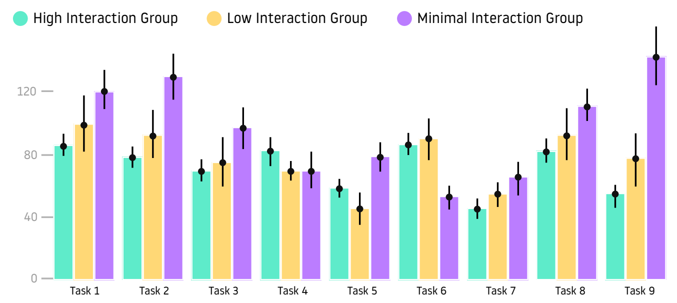


Fig. 17. Study 2 task completion time (secs.) with 95% confidence intervals.

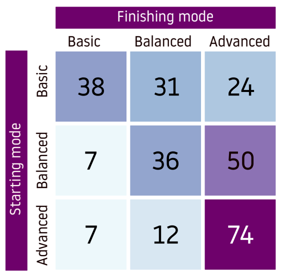


Fig. 18. Study 2 mode switches by starting mode. Users overwhelmingly switched to the more sophisticated interface mode.

starting mode and finishing mode of each task and each participant in a correlation matrix (Fig. 18). This observation suggests two characteristics of using multi-mode debugging tools. First, to answer **RQ2.2** programmers are roughly splitted 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.

C. Limitations

One threat to the validity of the evaluation is the number of participants. Although for each study we received hundreds of online participants, 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.

V. Discussion

This paper 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 will discuss a few interpretations of the results.

A. 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 [31], [32] 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.

B. 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.

C. 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

```haskell
f z
    | z == 3 = False
    | z == '4' = True
```

Listing 2. ChameleonIDE reports an error in the expressions `f` and `z`

the program shown in Listing 2, 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. And 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 [33] suggested this approach may provide a positive effect in helping programmers manage cognitive load and free up working-memory space for high-level thinking.

VI. Related Work

A. Finding all type error locations

Many have studied the approach of finding all locations that contribute to a type error [23], [26], [34], [35]. Type error slicing [23] 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 [26] 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 [20]. To the best of our knowledge, ours is the first user-centered evaluation of an interactive type debugging system involving type-error slicing.

B. Producing high-quality error explanation

One weakness of compiler error messages, in general, is that they fail to explain the error in human language. As put in [36], "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 [37] that ECEM has a positive result in understanding compiler errors. Decaf [38] 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 [39] 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 were not targeting type errors alone but general compiler errors (some even include runtime errors). The nuances of type errors, such as alternative typing, were not considered. Moreover, these explanation systems were designed specifically for novice users.

C. 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 [40] 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 [41] 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.

VII. Conclusion

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.

Acknowledgments

The work of Peter Stuckey was partially supported by the OPTIMA ARC ITTC, Project ID IC200100009.

References

[1] R. Chatley, A. Donaldson, and A. Mycroft, "The next 7000 programming languages," in *Computing and Software Science: State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. Woeginger, Eds. Springer International Publishing, 2019, pp. 250–282. [Online]. Available: https://doi.org/10.1007/978-3-319-91908-9_15

[2] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 153–162, ISSN: 1092-8138.

[3] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. Association for Computing Machinery, 2012, pp. 683–702. [Online]. Available: https://doi.org/10.1145/2384616.2384666

[4] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in JavaScript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 758–769, ISSN: 1558-1225.

[5] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," vol. 60, no. 10, pp. 91–100, 2017. [Online]. Available: https://doi.org/10.1145/3126905

[6] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ser. OOPSLA '13. Association for Computing Machinery, 2013, pp. 1–18. [Online]. Available: https://doi.org/10.1145/2509136.2509515

[7] Z. Chen, Y. Li, B. Chen, W. Ma, L. Chen, and B. Xu, "An empirical study on dynamic typing related practices in python systems," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. Association for Computing Machinery, 2020, pp. 83–93. [Online]. Available: https://doi.org/10.1145/3387904.3389253

[8] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of python dynamic features on change-proneness," 2015, pp. 134–139.

[9] Z. Xu, P. Liu, X. Zhang, and B. Xu, "Python predictive analysis for bug detection," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. Association for Computing Machinery, 2016, pp. 121–132. [Online]. Available: https://doi.org/10.1145/2950290.2950357

[10] Microsoft. JavaScript with syntax for types. [Online]. Available: https://www.typescriptlang.org/

[11] mypy. mypy - optional static typing for python. [Online]. Available: http://mypy-lang.org/

[12] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. Association for Computing Machinery, 2007, pp. 12–1–12–55. [Online]. Available: https://doi.org/10.1145/1238844.1238856

[13] Bill Wagner. (2022) Constraints on type parameters - c# programming guide. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

[14] Oracle. (2022) Generic methods and bounded type parameters (the java™ tutorials > learning the java language > generics (updated)). [Online]. Available: https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html

[15] Microsoft. (2022) Documentation - generics. [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/generics.html

[16] V. Tirronen, S. Uusi-Mäkelä, and V. Isomöttönen, "Understanding beginners' mistakes with haskell," vol. 25, p. e11, 2015, publisher: Cambridge University Press.

[17] J. Hage, "Solved and open problems in type error diagnosis," p. 13, 2020.

[18] B. Wu and S. Chen, "How type errors were fixed and what students did?" vol. 1, pp. 105:1–105:27, 2017. [Online]. Available: https://doi.org/10.1145/3133929

[19] Fu, Shuai. (2022) Chameleon type debugger. [Online]. Available: https://chameleon.typecheck.me/

[20] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells, "Improved type error reporting," in *In Proceedings of 12th International Workshop on Implementation of Functional Languages*. Citeseer, 2000.

[21] Microsoft. Visual studio code - code editing. redefined. [Online]. Available: https://code.visualstudio.com/

[22] Haskell. Haskell extension for visual studio code. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=haskell.haskell

[23] C. Haack and J. B. Wells, "Type error slicing in implicitly typed higher-order languages," vol. 50, no. 1, pp. 189–224, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016764230400005X

[24] Jakob Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.

[25] Ben Gamari. (2022) Home — the glasgow haskell compiler. [Online]. Available: https://www.haskell.org/ghc/

[26] P. J. Stuckey, M. Sulzmann, and J. Wazny, "Interactive type debugging in haskell," in *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '03*. ACM Press, 2003, pp. 72–83. [Online]. Available: http://portal.acm.org/citation.cfm?doid=871895.871903

[27] Y. Jun, G. Michaelson, and P. Trinder, "Explaining polymorphic types," vol. 45, pp. 436–452, 2002.

[28] OpenReplay. (2022) OpenReplay: Open-source session replay. [Online]. Available: https://openreplay.com/

[29] Github. (2022) GitHub topic: Haskell. [Online]. Available: https://github.com/topics/haskell

[30] Fu, Shuai and Dwyer, Tim and Stuckey, Peter, "Interactive haskell type inference exploration (extended abstract)," 2021. [Online]. Available: https://icfp21.sigplan.org/details/TyDe-2021/6/Interactive-Haskell-Type-Inference-Exploration-Extended-Abstract-

[31] A. Jbara and D. G. Feitelson, "How programmers read regular code: A controlled experiment using eye tracking," in *2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 244–254, ISSN: 1092-8138.

[32] N. Peitek, J. Siegmund, and S. Apel, "What drives the reading order of programmers?: An eye tracking study," in *Proceedings of the 28th International Conference on Program Comprehension*. ACM, 2020, pp. 342–353. [Online]. Available: https://dl.acm.org/doi/10.1145/3387904.3389279

[33] W. Crichton, M. Agrawala, and P. Hanrahan, "The role of working memory in program tracing," 2021. [Online]. Available: http://arxiv.org/abs/2101.06305

[34] Z. Pavlinovic, T. King, and T. Wies, "Practical SMT-based type error localization," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. Association for Computing Machinery, 2015, pp. 412–423. [Online]. Available: https://doi.org/10.1145/2784731.2784765

[35] T. Schilling, "Constraint-free type error slicing," in *Trends in Functional Programming*, ser. Lecture Notes in Computer Science, R. Peña and R. Page, Eds. Springer, 2012, pp. 1–16.

[36] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 575–585. [Online]. Available: https://doi.org/10.1109/ICSE.2017.59

[37] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen, "On novices' interaction with compiler error messages: A human factors approach," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. Association for Computing Machinery, 2017, pp. 74–82. [Online]. Available: https://doi.org/10.1145/3105726.3106169

[38] B. A. Becker, "An effective approach to enhancing compiler error messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 126–131. [Online]. Available: https://dl.acm.org/doi/10.1145/2839509.2844584

[39] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 633–643.

[40] H. Potter and C. Omar, "Hazel tutor: Guiding novices through type-driven development strategies," 2020, p. 10.

[41] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. Association for Computing Machinery, 2009, pp. 1569–1578. [Online]. Available: https://doi.org/10.1145/1518701.1518942