Interactive Haskell Type Inference Exploration

Extended Abstract

Shuai Fu
Monash University

Tim Dwyer
Monash University

Peter J. Stuckey
Monash University

ACM Reference Format:
Shuai Fu, Tim Dwyer, and Peter J. Stuckey. 2021. Interactive Haskell Type Inference Exploration: Extended Abstract. In *Proceedings of Type-Driven Development (TyDe'21).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In the last few decades, we have witnessed many significant advances in type-driven development. With type systems supporting dependent types and linear types, programmers may compose complex processes and structures purely in types. While strong types are quite successful at stopping erroneous programs from compiling, programmers can struggle to navigate through modern type systems.

Modern Integrated Development Environments (IDEs) can inform programmers of various information related to the source code, such as program state, execution results, and test status [Sulír et al., 2018]. However, when describing types and relations between types, especially when the program is in a state of type error, the IDE support is often minimal. To understand the type errors, programmers are often stuck with the traditional compile-time error messages in the form of a red squiggly line or multiple pages of text inside a terminal emulator. Studies find that text-based compiler error messages are not effective for hunting down issues in code [Barik et al., 2017, Becker et al., 2019].

We are currently developing and evaluating experimental features for IDEs that we believe can help novice programmers investigate type errors in the Haskell language.

2 The chameleon system

The chameleon system is a constraint-based type system and debugging interface for the Haskell language. An earlier version of the chameleon system was developed in the mid 2000s [Wazny, 2006]. It was conceived as a command-line tool rather than integrated into an editor or IDE. We have updated the chameleon system to use modern Haskell libraries and replaced the legacy C++ type unification algorithm with a Haskell implementation. We have also extended the Haskell parser and error reporting heuristics to make it suitable for use as a compiler service in a modern IDE (VS Code). Although constraint-based type systems have not yet gained a large audience, we find many of their advantages (e.g., better localization and traceability of type errors) can be amplified in the context of a graphic user interface. The chameleon system consists of two parts: the chameleon type debugger (2.1) and the interactive chameleon window (2.2).

2.1 Chameleon type debugger

The chameleon type debugger generates Constraint Handling Rules (CHRs) [Fruhwirth, 2009] and constraints based on reading the source program. It then consults the CHR solver to determine whether a term or program can be properly typed. One novel idea from the original chameleon study is, when it fails to compute the type of a specific term, we try to reduce the constraint store to a minimal unsatisfiable subset [Stuckey et al., 2003, Wazny, 2006]. This minimal unsatisfiable subset narrows down the problem surface while maintaining the locality of the error origin. The original chameleon type debugger was developed and published in 2007. After the publication, the development stalled until 2020 – when we started work on the improved chameleon system. Our work on this front focuses on transforming the minimal unsatisfiable subset into type error analysis.

2.2 Interactive chameleon window

The interactive chameleon window is a type system user interface and a set of tools we designed to complement the chameleon type debugger. Acknowledging the limitations of the text-based compiler error messages, we focus on error message enhancement for type inference [Becker et al., 2019]. The techniques we employed include text rephrasing, in-situ visualization, and interactive design. We present here three features in the interactive chameleon window that can work independently or in combination. These features are initially designed to teach and learn the Haskell language, however, we believe some features can benefit advanced Haskell users.

Type compare view. In a type error, the type compare view lists all (typically two) alternative ways a term can be typed

TyDe'21, Sun 22 – Fri 27 August 2021, Virtual Event
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

by ignoring one or more type rules. It then highlights the relative places in the source code that contribute to this alternative. The type compare view (see Fig. 1) does not make biased assumptions about what the programmer thinks is the proper type for a term. Instead of assuming a 'canonical type' and a 'wrong type', we prompt the question to the programmer to resolve.

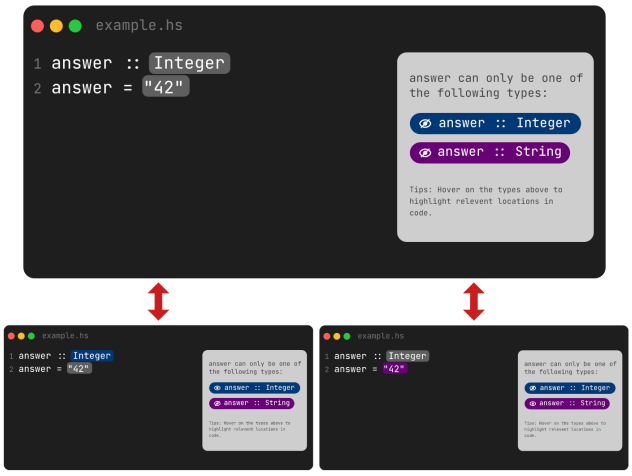

Figure 1. A type error in chameleon type compare view. In this case, the type error may be resolved in one of two ways, either by changing the type definition of the variable to `String`, or assigning the variable an `Integer` value. Chameleon presents both possibilities as equal candidates. Hovering over the types in the info pane (bottom-left and bottom-right) highlights the corresponding code with the matching colour.

Type Deduction View. We developed the type deduction view to illustrate better the minimal unsatisfiable subset and to unpack the type errors it entails. The type deduction view consists of many deduction steps. Each deduction step is a constraint inside the minimal unsatisfiable subset. The deduction steps are presented in the order in which unification takes place. We link each deduction step to the relevant locations of source program. When the programmer inspect one deduction step, the related locations are highlighted, and the highlights change as the programmer navigate to the next deduction step. From programmers' perspective, it is like breaking the type check into many small type checks, which programmers can easily reason about and verify. The challenge with the type deduction view is how to lower the learning curve of the various UI elements and build muscle memory for the interactive debugging process. For this, we find using a mini-map that mirrors the in-text highlights as step thumbnails is one of the more intuitive designs (see Fig. 2).

Figure 2. A sequence of screenshots of in type deduction view. The programmer can click on each deduction step to activate it. When activated, a deduction step is displayed in light color, and relevant location(s) in this deduction step are highlighted in the source code window.

Traffic Light Notation. Traffic light notation is a pictorial view (see Fig. 3) of the Haskell type language. Traffic light notation uses a dot to indicate simple types (e.g., `Int`) and uses dot(s) in a rectangular box to indicate types with a type constructor (e.g., `Maybe Int`). A similar idea was exercised [Jung and Michaelson, 2000] and proven helpful in this context.

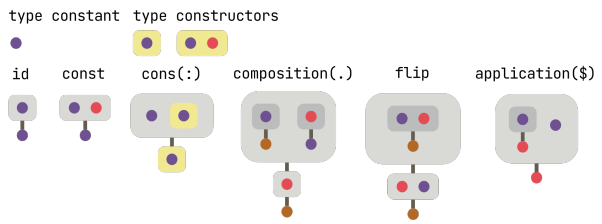

Figure 3. A list of common function types in traffic light notation. We use slight different shades of the same color to distinguish nested types like `[[a]]`.

Note that the traffic light notation does not replace the text-based type representation but instead offers a succinct view when the textual counterpart starts to feel unwieldy. Some examples include when the textual type signature is limited by its vocabulary (see Fig. 4) , when showing less information is helpful for beginners (see Fig. 5), or when the type itself is monstrous (see Fig. 6).

(a) The GHC output for an error involving type parameter.

```
Prelude> x = head id

<interactive>:8:10: error:
    • Couldn't match expected type '[a]' with actual type 'a0 -> a0'
    • Probable cause: 'id' is applied to too few arguments
      In the first argument of 'head', namely 'id'
      In the expression: head id
      In an equation for 'x': x = head id
    • Relevant bindings include x :: a (bound at <interactive>:8:1)
Prelude>
```

(b) The chameleon output for the same type error.

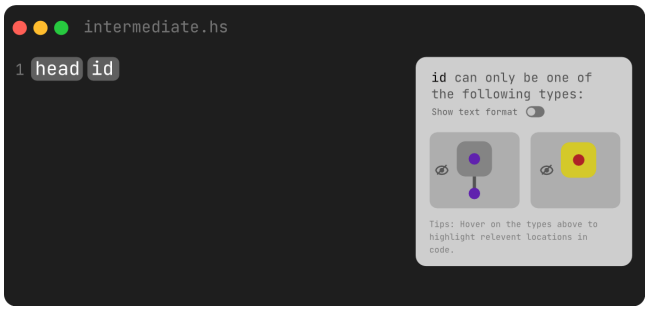

Figure 4. GHC may rename some type parameters during the type inference. This behaviour often adds cognitive overhead to understand the error message. In traffic light notation, programmers can quickly identify that the term `id` cannot be both an unary function type and a list type. The fact that both types are polymorphic with a type parameter 'a' does not obscure the real error.

3 Work in progress

Many advantages of the outlined features are currently speculations by the authors. While we are ourselves Haskell programmers, we do not speak for the community. We are in the process of running empirical studies to test how real Haskell users perceive our innovations on real-world programming tasks.

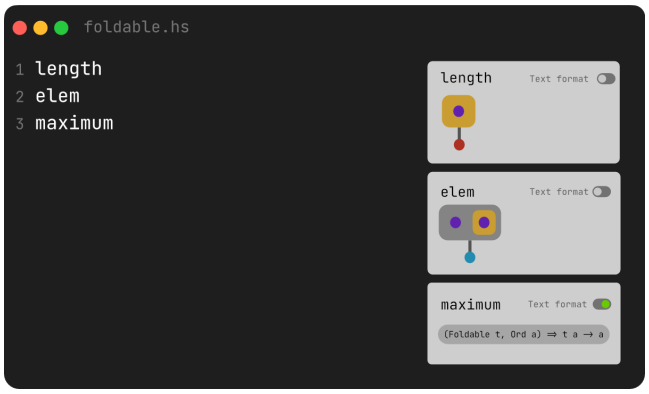

Figure 5. Many teachers teach Haskell using an alternative `Prelude` module to avoid explaining the `Foldable` type classes to beginners early on. In traffic light notation a list type and a type with a `Foldable` instance show the same shape. It allows teachers to teach fundamental concepts of functional programming without hiding the high level abstractions. The same goes for polymorphic numbers and strings.

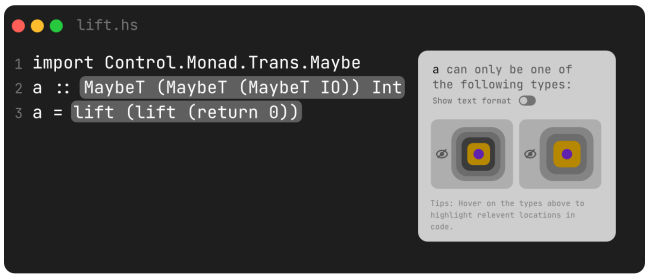

Figure 6. Debugging nested monadic code can frustrate even Haskell veterans. Viewing from the lens of traffic light notation, the error can be surprisingly clear.

The chameleon system is designed and implemented for Haskell 2010 standard without language extensions. We are working on the support for common type extensions, such as Generalised Algebraic Data Types (GADTs) and existential types. While studies proposed and developed constraint-based type systems that support various type extensions[Sulzmann et al., 2007, Wazny, 2006], our project largely interests in the usability and human-computer interactions. For us, the challenge of bringing more type features into play is how we communicate the ideas in a meaningful way and guide the programmers efficiently out of type dilemmas.

References

Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017-05). 575–585. https://doi.org/10.1109/ICSE.2017.59 ISSN: 1558-1225.

Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen Scotland Uk, 2019-12-18). ACM, 177–210. https://doi.org/10.1145/3344429.3372508

Thom Fruhwirth. 2009. Constraint Handling Rules. (2009). Cambridge University Press.

Yang Jung and Greg Michaelson. 2000. A visualisation of polymorphic type checking. 10, 1 (2000), 57–75. https://doi.org/10.1017/S0956796899003597

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '03* (Uppsala, Sweden, 2003). ACM Press, 72–83. https://doi.org/10.1145/871895.871903

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. 2007. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming* 17, 1 (2007), 83–129. https://doi.org/10.1017/S0956796806006137

Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

Jeremy Richard Wazny. 2006. Type inference and type error diagnosis for Hindley/Milner with extensions. University of Melbourne, PhD Thesis.