Usability of Error Messages in Liquid Types

CATARINA GAMBOA, 49535, Faculdade de Ciências da Universidade de Lisboa, Portugal

Error messages represent a fundamental interaction between developers and the programming language they use to implement programs. However, error messages are often confusing and difficult to understand, not helping the developer in the development process. The research on usability of compile error messages is scarce in traditional type systems implemented in languages such as Java or C. However, the research is even fewer in advanced type systems with liquid and dependent types. This essay announces the challenges of writing understandable error messages in liquid type systems and proposes an approach to improve them. The proposed approach incorporates the presentation of the error messages in a textual and a visual format to enhance their usability. Moreover, a methodology is proposed to evaluate the improved error messages.

CCS Concepts: • **[CTP] Deductive Program Verification**; • **[SI] User eXperience**;

Additional Key Words and Phrases: Liquid Types, Usability, Error Messages, Software Verification, Human-Computer Interaction

ACM Reference Format:
Catarina Gamboa. 2018. Usability of Error Messages in Liquid Types. In *Research Topics 2021-2022, February 2022*. Portugal, 9 pages.
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Compile error messages are an essential part of software development since they communicate to the developer the problems in the implementation that prevent the correct execution of the program. Furthermore, they provide fundamental feedback to developers and the main source of information in the implementation cycle since they depend on them to be able to fix the incorrect code before continuing with the implementation. Therefore, error messages should be seen as a fundamental part of the interaction between computers and developers.

However, it is common knowledge among developers that error messages are usually confusing and difficult to understand, and multiple studies on the subject make evident that frequently error messages fail to convey the intended information to developers [8, 26]. Moreover, poor error messages introduce significant barriers to novice programmers that depend on the given feedback advance in the implementation [7, 33]. Although, experts also have an interest in error messages [31], since they can behave as beginners again when they try new features of a language or change to an unfamiliar programming language [30].

Studies on enhancing error messages in languages with traditional type systems, such as Java or C, have shown improvements in the understanding and usability of the languages, and authors gathered guidelines to improve the text of error messages [13, 15, 24]. Additionally, there is an increase in the readability of error messages when they are accompanied with a suitable visual design, whether it is only a change on the format of the textual message [14] or the integration of messages in visual interfaces of development editors [9].

Notwithstanding, the usability of error messages has not been a topic of research in languages with advanced type systems with liquid and dependent types, where compile errors are usually more difficult to decipher and understand

Manuscript submitted to ACM

than in traditional languages. Therefore, this essay presents an approach to improve error messages in liquid and dependent types by focusing on enhancing the error messages in LiquidJava [18], an implementation of liquid types in the Java language.

2 BACKGROUND AND RELATED WORK

Type systems integrated into programming languages are one of the most popular methods for establishing guarantees about the behaviour of a program [21], allowing developers to specify the expected type of operations and verify the type before the program is executed. Dependent and liquid types advance traditional type systems by enabling the introduction of domain-specific information to a program via refinements to the type system. Liquid types [28] allow the addition of logical predicates to the basic types of a language, further restricting the types of the operations. In the context of liquid types, it is possible to encode dependencies between different terms and, therefore, have dependent types. Listing 1 illustrates the use of liquid and dependent types within the LiquidJava language. The example contains a method with two parameters, each with a refinement that specifies the expected type of the arguments when the method is invoked. Both annotations represent liquid types, the first parameter contains a liquid type that limits the value of `a` to a positive number, and the refinement in the second parameter represents a dependent type that indicates that `b`, the second parameter, needs to be greater or equal than the first parameter. The verification of liquid types is automatically performed during type checking, where verification conditions that were saved into the context are gathered and sent to an SMT-Solver that verifies their satisfiability.

```
1 void method-example(@Refinement("a > 0") int a, @Refinement("b >= a") int b);
```

Listing 1. Example of Liquid and Dependent Types.

Liquid types have been introduced as extensions to languages such as ML [17], Haskell [32], Javascript [12] and Scala [29], among others, but none of these implementations paid special attention to the error messages shown to developers. LiquidJava, a previous work by the author, tried to improve the error feedback delivered to users by giving detailed information and integrating the error reporting into Visual Studio Code [2], a popular IDE (Integrated Development Editor) amongst Java developers [4, 5]. However, during the user study conducted for the evaluation of LiquidJava, developers reported that they enjoyed the error reporting but found the error messages difficult to understand. Within the study, the participants were asked to report what they disliked about LiquidJava (in an open-ended question), and 10% reported that the error messages were not straightforward and should be improved [19]. These comments show that the previous efforts were not enough to make error messages useful to developers and present the current essay's motivation.

In dependently-typed languages, Eremondi et al. [16] claimed to present the first framework to improve error messages by applying heuristics in constraint graphs that capture the language dependencies. Their approach was applied in two popular dependently-typed languages, Agda [27] and Idris [10] and was able to produce errors in functions invocations that had the wrong number or incorrect order of arguments. However, the evaluation of their approach does not validate that the work improved the messages since they do not provide metrics to compare previous and improved messages or conduct an empirical evaluation with users.

Despite the scarce efforts in improving error messages in liquid and dependent type systems, other languages with advanced type systems have focused on the usability of error messages. One of these languages is Rust [3], a language designed to improve the reliability and efficiency of software that has a group within the language development team

only focused on improving error messages. Improvements in the messages reported by the team include the separation of information in sections and their customization with filters and colour highlights [25], the use of suggestion using developers' implemented code [11] and the use of extensible help with explanatory messages [1]. Another language recognized by its error messages is Elm [6, 22], which targets the development of reliable web applications and follows similar ideas to the ones presented in Rust. Both languages mostly focus on the error messages presented to users in terminals and not within editors, and the ideas behind the enhancement of the messages match guidelines provided by other studies.

In 2019, Becker et al. [7] present a landscape of the research on compiler error messages since 1965 and compiled ten guidelines to enhance text error messages; these include: providing examples and hints, using a positive tone and providing context to the errors. In 2021, Denny et al. [13] also proposed principles to improve error messages, in this case, based on an empirical study. The principles that differentiated the best from the worst error messages included the use of simple vocabulary (without jargon), messages presented in complete sentences, and the use of *economy of words*, meaning that the messages are as short as possible without sacrificing their intended meaning. The guidelines of both studies can be applied to error messages of any programming language given their broad context but do not include advice for dealing with error messages in more complex ecosystems.

3 MOTIVATIONAL EXAMPLE

Error messages in liquid and dependent type systems present different challenges from those in traditional strong type systems. The elemental example presented in listing 2 and the error message shown for the code (listing 3) showcase the main challenges that arise in creating understandable error messages in liquid types.

The example represents a method named `inRange` that takes two parameters. The first parameter has no refinement, which means that it uses the default refinement of `true`. In contrast, the second parameter has a refinement that ensures that its value has to be greater or equal to the value of the first parameter (`b >= a`). Additionally, the method also refines the return type by adding the annotation `@Refinement` on top of the method signature, and the refinement says that the return value needs to have a value between both parameters (`_ >= a && _ <= b`). However, within the method implementation, the return value does not fulfil the return refinement, and the error in listing 3 is displayed. In this example, the return value is not within the range of `a` and `b` because `a + 1` might not be smaller or equal to b, a counter-example that shows the error could be the invocation `inRange(0,0)` that would return the value 1, that is not smaller or equal to the second argument.

```
1 @Refinement("_ >= a && _ <= b")
2 public static int inRange(int a, @Refinement("b >= a") int b){
3     return a + 1; //Error
4 }
```

Listing 2. Implementation error in the return value according to the specification.

The error message includes the type that was expected and not found (line 2), the instruction that failed in the verification (line 5) and its localization in the code (line 24), the refinement found (lines 8 to 13), and the instance translation table (15-23). To understand why the error is showing in the return value, the developer has to check what is in the context and which of the verification conditions sent to the SMT-Solver prevented the success of the verification.

From the example, we can identify challenges that should be addressed in the presentation of error messages.

```
Refinement Type Error
Type expected:(#ret_21 >= a && #ret_21 <= b)
-------------------------------------------------
Failed to check refinement at:
return a + 1

Type expected: (#ret_21 >= a && #ret_21 <= b)
Refinement found:
∀#ret_21:int, (#ret_21 == #a_20 + (1)) =>
∀b:int, (b >= a) =>
∀#a_20:int, (#a_20 == a) =>
∀a:int, true =>
(#ret_21 >= a && #ret_21 <= b)

Instance translation table:
-------------------------------------------------
| Variable Name | Created in | File
-------------------------------------------------
| a | int a | Try.java:8, 35
| b | int b | Try.java:8, 64
| #a_20 | a | Try.java:9, 16
| #ret_21 | return a + 1 | Try.java:9, 9
-------------------------------------------------
Location: (/$PATH$/Try.java:9)
```

Listing 3. Error message produced from the example of listing 2.

Displaying the internal representation

Right at the beginning of the error message, the expected type shown to the user does not correspond to any code they have written in the implementation, which might be disconcerting to the developer. In this case, the return value previously written with an underscore was substituted by the value returned in the method implementation but, instead of substituting the `_` by `a + 1`, it is substituted by the variable `#ret_21`, that only exists in the internal representation of the program. The liquid type checking algorithm relies heavily on rewrites [21], changing operations to the Administrative Normal Form (ANF), renaming instances and function calls and introducing fresh variables that are never mentioned in the code (e.g., as path conditions). Although these changes are crucial to the internal verification, developers should not need to understand the inner transformations of the code to understand the message and change their own implementation.

Selecting context information

The information in context is used inside the liquid type verification. The context saves all the information on the refinements that were introduced in the code and the rewrites that introduced new variables (e.g., `#ret_21`). Therefore, the context information is highly important to understand the reason behind the errors. However, the context is shown in the error message in the section *Refinement found* and their creation in the *Instance translation table*) is not straightforward for users to understand. From the displayed information, the developer has to select what is relevant and discard redundancy cases. For example, in the error message above the variables in context `a:int, true`, and `#a_20:int, (#a_20 == a)`) represent non-valuable information to the user despite being important to the verification process.

Simulation of logic implications

The section of the message that represents the *Refinement found* includes the logical implications sent to the SMT-Solver, and that prevented the correct verification of the code. The best way for users to understand what triggered the error is to understand these implications by simulating them. This means that the user is almost doing the same work of the SMT Solver just to understand the problem and what they need to change in the code.

4 APPROACH

The approach proposed in this essay aims to improve the error messages in liquid types by proposing improvements in LiquidJava. This approach combines improving the text of the error message and also its presentation within a visual environment since both strategies have been proven useful in previous studies [20].

Inside an IDE, the error reporting usually happens with a red squiggle line below the conflicted code denoting the localization of the error. The red squiggle calls for the user attention, and only after this the user reads the error message. Therefore it is vital to use mechanisms that call for users' attention to the location of the error, and it is imperative that the error location is as specific as possible. Sometimes the red squiggle is even enough for users to identify the error and fix it. However, if that does not happen, the user will try to read the message. Although, if the message is very detailed, the users might give up on understanding it and focus again on guessing the error without using the provided message. Therefore, to allow different levels of detail, the first decision is to split the error message into two parts, one that represents the overview of the problem and can be shown along with the code (for example, on hovering the error), and the other part complements the overview with details that developers expand if necessary. The details are displayed in a separate editor tab customized to contain the necessary information, similarly to the custom tab of the Lean theorem-prover in Visual Studio Code [23].

Error Overview

The first message shown to users should give an overview of the error without getting into details. However, the terms that appear in the message must be familiar to the user, including using code that was implemented instead of using its internal representation. It also includes using terms that are not jargon to the developer. For the example of listing 2 one possible error overview could be represented as *"The return value* `(a+1)` *does not respect the return refinement, since it was not possible verify that* `(a+1)>= a && (a+1)<= b`".

To this end, the overview message can focus on the failed refinement and connect it to the code implemented by the users.

Extensible Details

The error details that appeared before appeared within the error message are now reassigned to a different view. The custom view of the details enables the user's interaction with the information so that they can dive into the details of different parts if wanted.

As seen in section 3, the context information is crucial to understand the errors that arise in the verification. Therefore, showing the refinement found and the context used in the details is essential. However, the current presentation of the refinement can be too detailed. Therefore, the refinement itself can be split into different levels of detail. The details can start with the type expected and present the refinements in context to the variables that appear in the expression. Then within each refinement, if new variables appear, the user can expand the refinements of the new variables. This

way, users do not see all the relevant context at once but can expand it if desired. Moreover, to decrease the redundant information in this section, it is possible to substitute the value of variables with simple refinements by the refinements themselves. For the example of listing 3, the predicate `(#a_20 == a)` could be simply substituted by `#a_20 == true` since `a` has the refinement of `true`.

In the previous messages, the *Instance translation table* aimed to match the internal representation of the variables with the place where they were created. However, since the table is decoupled from the refinements found, the information might not be as useful as it could. To bridge the two and to take advantage of having the code next to the custom tab, when hovering on the refinements found (in the custom tab), it is possible to highlight their place of creation in the code. Using the example, if the user hovers on the variable `b`, the refinement of the second parameter (on line 2 of listing 2) is highlighted.

In the custom view, besides the section of *Refinement found* is also a section to present a *Counter Example*. A counter-example for a failed verification can be given by the SMT Solver, where each variable used in the verification conditions is assigned a value that prevents the verification. The variables can then be arranged into a simple example given to the user. For the previous example of the `inRange` method, a simple counter-example that shows why the verification was not possible is `inRange(0,0)`, which, as explained in section 3 does not comply with the given specification.

Figure 1 is a mockup that represents the approach for the enhanced presentation of error messages in liquid types.

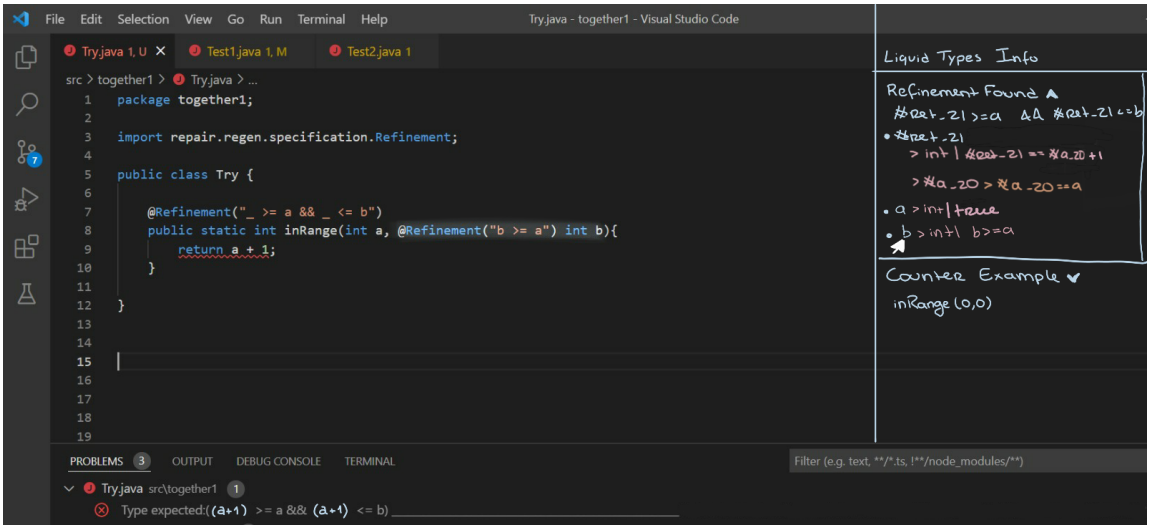

Fig. 1. Mockup representing approach.

5 METHODOLOGY

The proposed approach aims to improve error messages in liquid types so that developers can better understand the errors and fix the issues that appear when implementing code. Therefore, to evaluate the efficiency of the approach, it is essential to analyze how developers behave to enhanced messages compared to the previous ones. For that, an empirical user study can be performed.

The user study to evaluate this approach contains one main research question: "Do the changes in the error message presentation help developers find and fix implementation errors?". To answer this question, the user study focuses

on presenting programs with errors to participants with the previous version of the error message and the improved version. For the study, we aim to recruit 20 developers familiar with Java since LiquidJava is built upon Java, and this study does not aim to evaluate Java knowledge. Moreover, participants can have prior experience with liquid types, but it is not mandatory.

Since participants are not required to know about liquid types, the study starts with an introduction to liquid types within LiquidJava. This introduction uses a video and a website with examples[1], to reduce the bias of having a conductor of the study present the information that can change the behaviour between participants. After introducing the participants to the topic, they need to install the plugin for Visual Studio Code, where they will try to identify and fix implementation errors in six different programs. The first program is considered a trial and is not evaluated to allow developers to get used to the framework and prevent the first interaction from introducing erroneous values to the evaluation. After the trial program, five more programs are shown to the participants with an increasing degree of difficulty of the errors, which also means that more information will be available in the details. In each program, the participants need to report the line where the error is and what they propose to fix the issue. To allow us to compare the improvement of the error messages, half of the participants will use the previous version of the error messages, while the other half will have the enhanced version of the messages. This way, each participant only uses one version throughout the study, preventing the introduction of bias on changing from one version to the other.

During the study, the participants' answers will be collected, as well as the time each participant spent in each program for finding and fixing the bugs. The answers will then be classified in one of three categories, namely *Correct, Incorrect* or *Not Answered*. After gathering all data, we will compare the answers given for the previous error messages and the enhanced ones and compare the time developers took to finish each task in the two versions. Hence, we will be able to evaluate if the enhanced error messages positively impact the performance of developers.

6 CONCLUSION

Compiler error messages are essential in the interaction of developers with programming languages. The messages aim to give feedback that the developer needs to understand and use to modify the implemented code accordingly. However, the research on improving the usability of error messages is scarce and almost nonexistent when we move from traditional type systems to more advanced type systems that implement liquid and dependent types.

This essay shows the challenges of having understandable and useful errors in liquid types and proposes an approach for enhancing the error messages produced on the verification process of languages with liquid and dependent types. The examples and ideas in the essay are focused on LiquidJava, an implementation of liquid types in Java, to materialize the ideas in an existing language. The proposed approach changes the presentation of the error message to make the most of the textual and visual representation. Hence, the error message is split into a textual overview and a detailed view. The overview message uses the implementation terms and aims to give a high-level description of the error. Additionally, the detailed view aims to give extensible details related to the refinements found and their values in context and introduce a counter-example for the failed verification. Moreover, the user can match the variables' refinements to the implemented code by hovering on them on the detailed view to understand where the code's refinements appear.

This essay also presents a methodology for the evaluation of the proposed approach. Since the main goal of this work is to improve error messages to help developers understand the errors and fix them, the evaluation of the approach involves a user study where participants try to find and fix errors in programs with liquid types. Hence, to compare

[1] https://catarinagamboa.github.io/liquidjava.html

the previous version of the messages with the new, improved version, the participants are split into two groups and only use one of the error versions. Given the answers of the participants and the time they spent on the tasks, we can evaluate if the proposed approach helps developers have a better programming experience and better performance.

7 ACKNOWLEDGMENTS

This work was supported by FCT through the LASIGE Research Unit, references UIDB/00408/2020 and UIDP/00408/2020, and the project CAMELOT (POCI-01-0247-FEDER-045915), funded the ERDF - European Regional Development Fund (COMPETE 2020, CENTRO 2020, LISBOA 2020, FCT, CMU|Portugal).

REFERENCES

[1] 2016. Shape of errors to come: Rust blog. https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html
[2] 2016. Visual Studio Code. https://code.visualstudio.com/
[3] 2018. The rust programming language. https://doc.rust-lang.org/book/ch19-04-advanced-types.html
[4] 2021. 2021 Java technology report. https://www.jrebel.com/blog/2021-java-technology-report
[5] 2021. JVM ecosystem REPORT 2021. https://snyk.io/jvm-ecosystem-report-2021/
[6] 2022. Elm. https://elm-lang.org/
[7] 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 *2019 ITiCSE Working Group Reports* (Aberdeen, Scotland UK) *(ITiCSE-WGR '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3344429.3372508
[8] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, New York, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432
[9] Lorenzo Bettini. 2019. Type errors for the IDE with Xtext and Xsemantics. *Open Computer Science* 9, 1 (2019), 52–79. https://doi.org/10.1515/comp-2019-0003
[10] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.
[11] Sean Chen. 2020. The Anatomy of Error Messages in Rust — RustFest Global 2020. https://www.youtube.com/watch?v=oMskswu1SxM&list=PL85XCvVPmGQiudPknCxiSpybc5RTfkXe6
[12] Chugh R., Herman D., Jhala R. 2012. Dependent Types for JavaScript. http://goto.ucsd.edu/~ravi/research/oopsla12-djs.pdf.
[13] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C. Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker (Eds.). ACM, 55:1–55:15. https://doi.org/10.1145/3411764.3445696
[14] Tao Dong and Kandarp Khandwala. 2019. The Impact of "Cosmetic" Changes on the Usability of Error Messages. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–6. https://doi.org/10.1145/3290607.3312978
[15] Nabil El Boustani and Jurriaan Hage. 2011. Improving Type Error Messages for Generic Java. In *Higher-Order and Symbolic Computation*, Vol. 24. Savannah, GA, 3–39. https://doi.org/10.1007/s10990-011-9070-3
[16] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. 2019. A framework for improving error messages in dependently-typed languages. *Open Comput. Sci.* 9, 1 (2019), 1–32. https://doi.org/10.1515/comp-2019-0001
[17] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468
[18] Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. 2021. User-driven Design and Evaluation of Liquid Types in Java. *CoRR* abs/2110.05444 (2021). arXiv:2110.05444 https://arxiv.org/abs/2110.05444
[19] Catarina Gamboa, Paulo Canelas Santos, Christopher Timperley, and Alcides Fonseca. 2021. LiquidJava: Adding Lightweight Verification to Java. In *INFORUM*. –.
[20] Jesna AA and Renumol V. G. 2016. An IDE for Java with Multilevel Hints to Develop Debugging Skills in Novices. In *IAFOR International Conference on Education, IICEDubai2016*. 1–14. www.iafor.org
[21] Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *CoRR* abs/2010.07763 (2020). arXiv:2010.07763 https://arxiv.org/abs/2010.07763
[22] Charlie Koster. 2017. Advanced types in elm - opaque types. https://ckoster22.medium.com/advanced-types-in-elm-opaque-types-ec5ec3b84ed2
[23] Lean for VS Code 2021. Lean for VS Code. https://github.com/leanprover/vscode-lean

[24] Derrell Lipman. 2014. *LearnCS! a Browser-Based Research Platform for CS1 and Studying the Role of Instruction of Debugging from Early in the Course.* Ph. D. Dissertation. University of Massachusetts Lowell.

[25] Jane Lusby. 2020. Rustconf 2020 - error handling isn't all about errors. https://www.youtube.com/watch?v=rAF8mLI0naQ

[26] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[27] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science, Vol. 5832)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.). Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[28] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[29] Georg Stefan Schmid and Viktor Kuncak. 2016. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 31–40. https://doi.org/10.1145/2998392.2998398

[30] Jean Scholtz and Susan Wiedenbeck. 1993. Using Unfamiliar Programming Languages: The Effects on Expertise. *Interacting with Computers* 5, 1 (1993), 13–30.

[31] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010, Article 3 (Jan. 2010), 26 pages. https://doi.org/10.1155/2010/602570

[32] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

[33] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: Using Crowd-sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In *Proceedings of the 11th International Conference on Advances in Web-Based Learning* (Sinaia, Romania) *(ICWL'12)*. Springer-Verlag, Berlin, Heidelberg, 228–239. https://doi.org/10.1007/978-3-642-33642-3_25