Enhancing Proof Assistant Error Messages with Hints: A User Study

Master's Thesis

Maria Khakimova

Enhancing Proof Assistant Error Messages with Hints: A User Study

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Maria Khakimova

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: A drawing of Gellig, one of my partner's chickens.

Enhancing Proof Assistant Error Messages with Hints: A User Study

Author:	Maria Khakimova
Student id:	5277736

m.

Abstract

Proof assistants are tools that allow programmers to write formal proofs. However, despite the improvements made in recent years, there is still a limited number of published user studies in improving the usability of dependently-typed proof assistants. This is especially true for investigating the effects of *enhancing error messages* on novice programmers, even though it is a popular topic for imperative languages.

This thesis aimed to help fill this gap in knowledge by using Agda as a research vector. We implemented hint enhancements for the error messages displayed upon three common mistakes: forgetting whitespace, using confusable Unicode characters, and supplying too few arguments to a function. A between-participants user study was then conducted with 70 students to determine the effects that these error messages had on the usability of Agda.

Results showed statistically significant improvements in the number of compiling submissions, and the rated "helpfulness" of the error messages (when compared to the original messages without hints). However, we were not able to determine if there was any statistically significant impact on the overall speed at which students resolved the errors. Furthermore, while we identified decreases in the ratings of "incorrect" hints, they did not appear to significantly influence the success rate, or time taken to fix an error. We concluded that enhancing error messages with hints is a promising avenue for improving the usability of dependently-typed proof assistants.

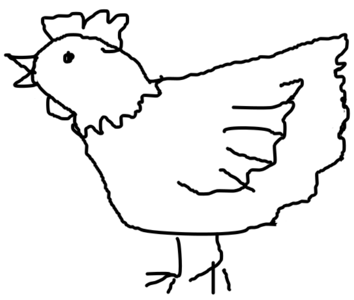
Preface

Somehow, I can't quite believe that this is the end of the long journey that was my education at TU Delft. Writing my thesis[1,2] has been so hectic that I haven't really been able to sit down and process that fact, but now that I'm writing this preface it's...huh.

Of course, there are many people that I would like to thank for helping me through all of this:

- my wonderful supervisors, Jesper Cockx, Sára Juhošová, and Jaro Reinders, who have supported me, and given much useful advice throughout this process,
- my partner, Eliott Afriat, for providing moral support whenever I asked for it (as well as introducing me to Toby, who is perhaps the best cat in the world), and most importantly
- my parents, for everything. Words cannot express just how much they have helped me throughout my entire education.

On a slightly different note, I would also like to thank this little chicken doodle that has kept me sane through writing all of my thesis drafts:


The chicken doodle that used to be on the cover page for this thesis

Despite being hectically sketched with a mouse, she has bought me much joy throughout the thesis process[3].

Maria Khakimova
Delft, the Netherlands
June 17, 2025

[1]And of course, still working on PrescriptIO to make sure that it was in a good state for another user study.

[2]I have just realised that I cannot add footnotes to footnotes without trickery, and I am devastated.

[3]While I did not choose this thesis topic simply because Agda was named after a chicken, I did enjoy the fact that it gave me the excuse to draw more chickens.

Contents

Preface **iii**

Contents **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
- 1.1 Dependently-Typed Proof Assistants . . . 1
- 1.2 Enhancing Error Messages . . . 3
- 1.3 Research Questions and Methodology . . . 3
- 1.4 Thesis Overview . . . 5

2 Existing Agda Hints **7**
- 2.1 Identifying Existing Hints in Agda . . . 7
- 2.2 Trends in Hint Structure . . . 8
- 2.3 How to Proceed With this Information . . . 11
- 2.4 Limitations . . . 11

3 Implementation **13**
- 3.1 General Design and Implementation Decisions . . . 13
- 3.2 Missing Space . . . 15
- 3.3 Unicode Confusables . . . 18
- 3.4 Too Few Args . . . 24

4 Experimental Setup **29**
- 4.1 Recruitment . . . 29
- 4.2 Deploying the Survey . . . 29
- 4.3 The Experiment . . . 30
- 4.4 Pilot Study . . . 33

5 Results **35**
- 5.1 Statistical Significance: The Mann-Whitney U Test . . . 35
- 5.2 Programming Question Response Rate . . . 36
- 5.3 Subjective Opinions of the Error Messages . . . 36
- 5.4 Novices' Performance . . . 41

6 Discussion **49**

6.1 What Does The Data Say? 49
6.2 Threats to Validity 52

7 Related Work **55**
7.1 Usability of Proof Assistants 55
7.2 What Errors do Novice Programmers Make? 56
7.3 Enhancing Error Messages 57
7.4 Improving Error Messages in Functional Languages 58

8 Conclusion **59**
8.1 Advice for User Studies 59
8.2 Further Work 60

Bibliography **63**

Acronyms **69**

A Full List of Identified Hints in Agda **71**
A.1 Directions 71
A.2 Suggestions 72
A.3 Questions 74

B User Study: Brightspace Announcement **75**

C User Study: Informed Consent **77**

D Results: "Longest Intervals" Longer than 2 Minutes **79**

List of Figures

2.1 The number of existing hints found per hint type 8
2.4 The distribution of hints found with and without framing phrases 10

4.3 The WebLab interface on a programming question 31
4.4 The Likert scale, as displayed in WebLab 32

5.1 The number of participants who received the survey variants 36
5.2 The total number of programming question submissions, those with timing data and those without 36
5.3 The distribution of the number of programming question submissions per question 36
5.4 Overview of novices' perceptions of the helpfulness of the error messages 37
5.6 Perceptions of the error message helpfulness: all submissions ("Correct" hints) . 38
5.8 Perceptions of the error message helpfulness: unsuccessful submissions ("Correct" hints) 39
5.10 Perceptions of the error message helpfulness: all submissions ("Incorrect" hints) 40
5.12 Perceptions of the error message helpfulness: unsuccessful submissions ("Incorrect" hints) 41
5.14 Aggregated number of responses per question 42
5.16 Number of responses per question ("Correct" hints) 43
5.18 Number of responses per question ("Incorrect" hints) 44
5.20 Box plot comparing the times spent per question ("Correct" hints) 46
5.23 Box plot comparing the times spent per question ("Incorrect" hints) 47

List of Tables

2.2 The distributions of the identified information orderings among different hint types 9
2.3 The distributions of the identified framing phrases among different hint types . . 9
2.5 The distributions of the identified structural presentations among different hint types 10

4.1 The lecture structure in CSE3100: Functional Programming 29
4.2 The five groups of questions used in the user study 31
4.5 The ten programming questions used in the user study 33

5.5 Aggregated results of the single-tailed MWU tests (perceived helpfulness) 37
5.7 Single-tailed MWU results: are "correct" hints more helpful (all submissions)? . 38
5.9 Single-tailed MWU results: are "correct" hints more helpful (unsuccessful submissions)? 39
5.11 Single-tailed MWU results: are "incorrect" hints more misleading (all submissions)? 40
5.13 Single-tailed MWU results: are "incorrect" hints more misleading (unsuccessful submissions)? 41
5.15 Aggregated results of the single-tailed MWU tests (success rate) 42
5.17 Single-tailed MWU results: do "correct" hints increase the success rate? 43
5.19 Single-tailed MWU results: do "incorrect" hints decrease the success rate? 44
5.21 Comparison of the mean and median times spent per question ("Correct" hints) 46
5.22 Single-tailed MWU results: do "correct" hints decrease the time spent? 46
5.24 Comparison of mean and median times spent per question ("Incorrect" hints) . 48
5.25 Single-tailed MWU results: do "incorrect" hints increase the time spent? 48

6.1 A comparison of how = (U+003D) and ᐀ (U+1400) are presented in WebLab . . 50

Chapter 1

Introduction

Encountering error messages is an unavoidable part of programming, with Java programmers spending 13%-25% of their time reading compiler error messages [1]. Error messages communicate information to developers, aiding them in understanding the roots of the errors and bringing them closer to a working program. This feedback is especially important to novice programmers, as they have limited experience with the language.

Unfortunately, error messages are known to cause significant frustration to novice programmers, and are frequently considered to be a "barrier to progress" [2]. Inscrutable or frustrating error messages can cause students to lose confidence [3] and potentially demotivate them from learning the language. Additionally, confusing error messages can lead developers "down the wrong path to correcting the problem" [4], which novices are most vulnerable to. Such error messages can also indirectly affect educators, forcing them to explain the same error message to multiple students [3].

When considering that novices do read error messages [5], and that the difficulty of reading error messages has been found to predict performance in programming tasks [1], it is clear that making error messages novice-friendly is an important task. To address this concern, multiple researchers have tried to use Human-Computer Interaction (HCI) guidelines to enhance error messages in imperative languages like Java, C++, and Python [6, 7, 8, 2, 9, 10, 11, 5, 12, 13, 14, 15]. However, there has been limited research into enhancing error messages provided in dependently-typed languages or proof assistants.

1.1 Dependently-Typed Proof Assistants

Dependent types are types which can "depend on elements of other types" [16], allowing for more expressive types, as well as assisting in compiler optimisation and error detection [17]. For example, in a dependently-typed language like Agda, we can define a dependent type `Vec A b` as follows[1]:

```
data Vec (A : Set a) : ℕ → Set a where
  []  : Vec A zero
  _::_ : ∀ (x : A) (xs : Vec A n) → Vec A (suc n)
```

Here, the type `Vec A n` would refer to a vector with elements of type `A` and a given length `n`. Using this definition, we can write functions like `append`, which has the following type signature verifying that the length of the final vector is correct:

[1]This definition of a vector comes from the Agda standard library: https://agda.github.io/agda-stdlib/v2.2/Data.Vec.Base.html

```
Vec A m → Vec A n → Vec A (m + n)
```

Furthermore, the expressive types permitted by a dependently typed system allow such languages to be used as proof assistants [18].

Proof assistants have successfully been applied in many projects. For example, the CompCert project used the dependently-typed Rocq Prover (formerly known as Coq) to implement the formally verified C compiler CompCert C [19]. This was done to prevent the risk of *miscompilation*, where a compiler inserts bugs into programs during compilation. Furthermore, ACL2[2] has been used in industrial projects by companies such as Motorola, AMD, Intel, and Oracle to verify both software and hardware [20].

1.1.1 What is Agda?

We have selected Agda as a research vehicle for investigating the effects of enhancing error messages in dependently-typed proof assistants. This was due to our familiarity with the language, as well as it being commonly used in education (particularly at TU Delft).

Agda is a dependently-typed functional programming language and proof assistant that is based on intuitionistic type theory [21]. It allows programmers to interactively work together with its compiler to produce formal proofs [22], be that abstract mathematical proofs, or the formal verification of software. The proofs that users can write about their programs in Agda can ensure that software is correct by construction, as opposed to writing tests which famously show the presence of bugs, but not their absence [23].

However, Agda, as well as other dependently-typed languages, has not experienced widespread usage [24], and is known to have a steep learning curve and to not be accessible to novice programmers [25, 26]. Researchers have attributed this to the mathematics-oriented design [25], the complexity of the underlying theories behind proof assistants [24, 26], and their experimental nature [24]. However, the quality of error messages also plays a role in novices' experience with Agda, with a recent user study finding that unclear error messages were the most prominent obstacle encountered by novice programmers [25].

In addition to complex type errors induced by the dependent type system, Agda has syntax features that are not frequently present in other languages. For example, students tend to find issues with the unconventional importance of whitespace [25]: given a mixfix operator `_,_`, `(a, b)` and `(a , b)` are not equivalent. This is because in the first case `a,` gets parsed as a variable name.

Additionally, Agda has Unicode support which is used in most library code. While it is not unique in this regard, with languages like APL [27], Julia [28], Raku [29], and the now-discontinued Fortress [30] also supporting non-standard Unicode, these are languages that the majority of novices in Agda are not likely to have learnt or worked with previously. This can lead programmers typing visually identical identifiers, such as `!` (U+0021) instead of `ǃ` (U+01C3), with one being in scope and the other not. This type of confusion has already been noted in many Rust projects [31].

While research has been done to improve error messages in a dependently-typed core calculus through modifying a higher-order unification algorithm with replay graphs [24], no user studies have been conducted to test if programmers prefer the new error messages over originals. Additionally, no studies have been conducted in enhancing error messages in dependently-typed languages by using modern HCI guidelines.

We believe that improving the usability of dependently-typed languages like Agda is vital for their widespread adoption. By improving the quality of error messages from a novice standpoint, we aim to make it a more enjoyable language to learn, increasing its usage and thereby the presence of formally verified software.

[2]ACL2 is a proof assistant based on first-order logic.

1.2 Enhancing Error Messages

In 2019, Becker *et al.* conducted a literature review on research into the effectiveness of error messages and their enhancements. Based on the results of their survey, they identified nine guidelines for designing and enhancing error messages:

- Increase Readability,
- Reduce Cognitive Load,
- Provide Context,
- Use a Positive Tone,
- Show Examples,
- Show Solutions or Hints,
- Allow Dynamic Interaction,
- Provide Scaffolding,
- Use Logical Argumentation, and
- Report Errors at the Right Time.

While there is a lot of empirical evidence supporting the enhancements of error messages [2, 9, 10, 11, 5, 32], there are several gaps identified in the literature:

- There is limited research that isolates the effect of *one* enhancement on the usability of error messages.
- None of the research that we were able to identify into improving or enhancing error messages in dependently-typed languages (including proof assistants) conducts a *user study* to empirically measure the effects of the improved error messages on users.

The focus of this study will thus be to enhance Agda's error messages with one recommendation from Becker *et al.*'s guidelines, and to evaluate the obtained enhanced error messages (EEMs) with a user study.

Out of the potential enhancements, we decided to provide hints or solutions to the programmer. Although there is a lot of empirical evidence that supports the positive impact of hint error message enhancements [7, 14, 6, 8], some researchers also warn against providing hints, with worries that they can lead users "down the wrong path" [4]. Furthermore, despite Becker *et al.*'s comprehensive literature review finding the greatest number of papers providing empirical evidence for showing solutions or hints, all of the studies combined the addition of hints with other enhancements. Therefore, it is difficult to isolate the effect that augmenting error messages with hints has on the usability of the error messages.

1.3 Research Questions and Methodology

The goal of this thesis is to investigate how hint-based enhancements to Agda's error messages help novices learn and use them more effectively. Hence, we investigate the following research questions:

- **RQ1** *Which kinds of hints or solutions provided in Agda's error messages are valued by novices?*
- **RQ2** *To what extent does providing novices with solutions or hints in Agda's error messages improve their ability to fix errors?*

RQ2 is further split into the following sub-questions:

- **RQ2.1** *How does enhancing Agda's error messages with hints influence the probability that novices are able to resolve the error?*

- **RQ2.2** *How does enhancing Agda's error messages with hints affect the time novices spend fixing the error?*

In order to analyse the usefulness of enhancing error messages with hints, it is necessary to have samples of such error messages. We could obtain these in three ways:

1. use existing Agda error messages that are already augmented with hints,
2. design "ideal" hints for error messages that do not have them, without implementing them in Agda, or
3. design hints for error messages that do not have them, and implement them as extensions to Agda.

As will be discussed in Chapter 2, there are already error messages incorporating hints that novices can encounter frequently. The helpfulness of these hints could thus be investigated by having two versions of Agda — the original version, and one with the hints removed. This type of study would allow us to analyse if more resources should be allocated to enhancing error messages in Agda and other dependently-typed languages. However, it would not help in advancing the quality of Agda's error messages. As such, there would be no tangible benefits to novice Agda users in the short-term. Most importantly, this would make the user study test the effect of *removing* hints instead of *adding* them, which might have different results. This would mean that the findings of such a user study would not directly support enhancing error messages with hints.

The second option was to create "ideal" error messages *without* implementing them in Agda. This was an appealing option because it would remove the restrictions imposed by the structure of the Agda compiler, as well as those placed by time limitations. We could have then studied their helpfulness by having the participants give a walkthrough of what they thought the error was, and how they would go about solving it.

However, this option was also dismissed, as it would not allow participants in the user study to respond to the error messages in a representative manner. People do not normally give cognitive walkthroughs while coding, and the process could influence how they approach fixing the code. For example, they might make fewer small exploratory changes. Furthermore, if the hint misled the participants, they would not encounter the next error message caused by their attempted "fix". Additionally, we wanted the enhanced error messages to be realistic within the confines of what the Agda compiler allowed. While this would mean that the quality of the hints would not reach the ceiling of what is possible, this would allow this thesis to give better advice for how to proceed with enhancing error messages in the future.

Thus, the final option was chosen, and the enhanced error messages were both designed and implemented in Agda. Although this would limit the number of error messages we would be able to enhance with hints and investigate, this method would allow the user study to collect the most representative data.

This thesis answers the research questions as follows:

1. We examine HCI literature to determine whether it supports including solutions or hints in error messages as a method of enhancing such messages for novice programmers.
2. We investigate the Agda codebase to see if Agda already implements hints or solutions in error messages, using the findings to make our hint additions as consistent with prior work as possible.
3. We select several error messages that lack hints, and enhance them with potential solutions/hints in the Agda codebase:

Hint	**Mistake Explanation**	**Error Type**
Missing Space	Forgetting whitespace between variables or operators	`[NotInScope]`
Unicode Confusables	Using the wrong visually confusable Unicode character	`[NotInScope]`
Too Few Args	Not enough arguments passed to a function	`[UnequalTerms]`

4. We evaluate the usefulness of the enhanced error messages by conducting a user study where novice Agda programmers are asked to fix errors in pre-written code and reflect on the enhanced error messages.

We provide the following contributions:

1. We provide an overview of the existing hints in Agda's error messages in Chapter 2.
2. We implement hint enhancements for the selected enhanced error messages (see Chapter 3), which are made available at
    - Missing Whitespace: `https://github.com/mkhakimova42/agda/pull/3`
    - Unicode Confusables: `https://github.com/mkhakimova42/agda/pull/2`
    - Too Few Args: `https://github.com/mkhakimova42/agda/pull/1`
3. We create a user study design, with recommendations for consecutive user studies in Chapter 4.
4. We provide a discussion with recommendations about if and how error messages should be enhanced with hints in Agda and similar dependently-typed languages in Chapter 6.

1.4 Thesis Overview

The thesis is structured as follows. In Chapter 2 we conduct a brief survey of error messages in Agda that are already augmented with hints, and discuss the trends in formatting these hints. Then, in Chapter 3 we discuss the error messages that we selected to enhance with hints, and how they were enhanced. This is followed up by Chapter 4, where we present the design of the user study, the results of which are displayed in Chapter 5. Chapter 6 provides a discussion of the results, as well as the potential threats to validity. We present a survey of the related work in Chapter 7, and give our conclusions, as well as advice for future work, in Chapter 8.

Chapter 2

Existing Agda Hints

One of the important principles for error message design is consistency in how error messages are presented (e.g. if error messages do not anthropomorphise the compiler[1], this rule should be applied to *all* error messages) [33]. This serves to reduce cognitive load on the programmers, allowing them to more efficiently process the information [3]. Work on enhancing error messages in Agda (in addition to all other languages) should thus strive to maintain as much consistency with existing error messages as possible.

The purpose of this section is to understand the current state of error message hint augmentations in Agda. We identified error messages that were augmented with hints from a subset of the Agda codebase, as described in Section 2.1. We then categorised the hints based on:

- the type of hint, and
- the structural presentation of the hint.

We analyse the patterns in these categories in Section 2.2, and provide advice for future work with hints in Agda's error messages in Section 2.3. Furthermore, we briefly discuss the limitations of this investigation in Section 2.4.

2.1 Identifying Existing Hints in Agda

To identify hints within existing error messages, we needed to determine what criteria to use to classify text within an error message as a "hint". For the purpose of this survey, we used the following criteria:

- A hint has to be an *addition* to the base error message.
- A hint needs to provide a (potential) solution to the programmer. This can either be done through *directing* the programmer to do something, or *asking* them if they "meant" or "wanted" to do something else.

Agda is a massive project, with many unique error messages — as an approximation, we identified 1692 test cases in the `test/Fail` directory that test error messages. A thorough investigation of *all* the hints that can be presented was thus out of scope. Therefore, we used heuristics to attempt to get as many existing hints as possible. Although we cannot guarantee that our list is comprehensive, it should still give an idea of the general patterns in how Agda's hints are structured.

In Agda, there are "error" and "warning" messages, both of which are relevant to our investigation. Many of Agda's warnings are classified as "error warnings": errors that only get

[1]The compiler can be anthropomorphised in sentences like "I give up!" or "I do not understand...".

displayed as warnings to allow type-checking to continue [34]. Furthermore, warning messages can be treated as errors with the use of flags [34]. Therefore, we looked for occurrences of both, combing over the modules where they are defined:

- `Agda.Interaction.Options.Errors`
- `Agda.Syntax.Concrete.Definitions.Errors`
- `Agda.TypeChecking.Errors`
- `Agda.TypeChecking.Serialise.Instances.Errors`
- `Agda.Interaction.Options.Warnings`
- `Agda.TypeChecking.Monad.Base.Warning`
- `Agda.TypeChecking.Warnings`
- `Agda.TypeChecking.Pretty.Warning`
- `Agda.Syntax.Parser.Monad`

Furthermore, since `parseErrors` are designed to have the error message passed to them, we also looked for all instances of `parseError` in the Agda project.

2.2 Trends in Hint Structure

The full list of the identified hints can be viewed in Appendix A. For the analysis of the results to be more specific to hint context, all the hints were categorised based on their *type*. The type of hint was categorised into three groups:

- Directions: directions to the programmer,
- Suggestions: tentative suggestions to the programmer, and
- Questions: questions directed at the programmer.

Figure 2.1 shows the distribution of the types among the recorded hints.

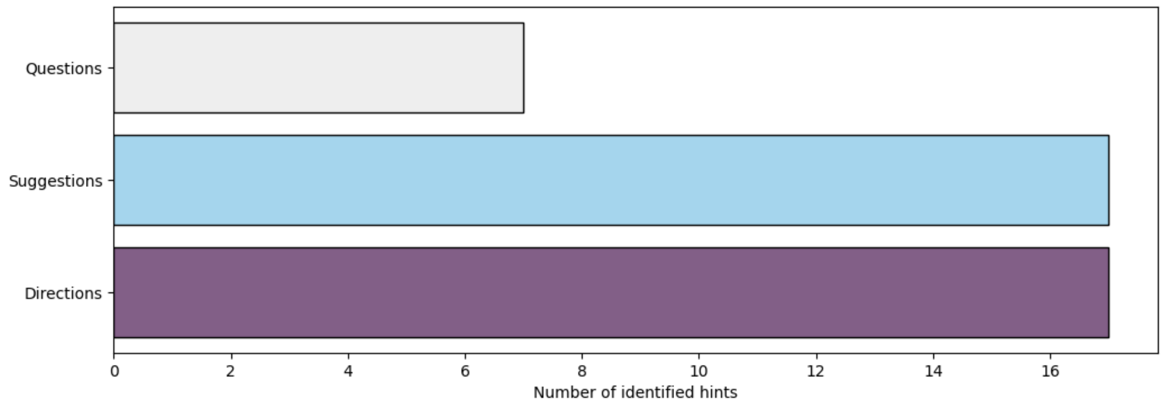


Figure 2.1: The number of existing hints found per hint type

This section looks at three aspects of the hints:

- the order of the presented information (see Section 2.2.1),
- the framing phrases that were used (see Section 2.2.2), and
- the structural presentation of the hints (see Section 2.2.3).

2.2.1 Information Order

One factor that differed between error message hints was the order in which information is presented. Error messages oscillating between different orders can potentially make it less clear where the programmer needs to look to get the information they are looking for. Therefore, we believed that keeping this order consistent is desirable.

We identified eight different orderings of information in the Agda hints. Table 2.2 shows the distribution of these information orderings among the different hint types.

Information Order	**Directions**	**Suggestions**	**Questions**	**Total**	**Percentage Use**
Instruction → Purpose	**10**	4	0	14	34%
Instruction → Purpose → Instruction	1	0	0	1	2%
Purpose → Instruction	2	0	0	2	5%
Purpose → Expectation	2	0	0	2	5%
Conditional → Instruction	2	0	0	2	5%
Instruction → Conditional	0	1	0	1	2%
Instruction	0	**10**	2	12	29%
Expectation	0	2	0	2	5%
Assumption	0	0	**5**	5	12%
Total	**17**	**17**	**7**	**41**	

Table 2.2: The distributions of the identified information orderings among different hint types

It can be seen that the information order is not consistent among the hints. "Instruction → Purpose" was the most common information order. However, it is only used 34% of the time. Just "Instruction" (29%) and "Assumption" (12%) should also be noted for being the most popular within the Suggestions and Questions categories respectively.

2.2.2 Framing Phrases

Another feature of the hints that we identified were framing phrases that encompassed or served as a preamble to the content of the hint. Keeping this usage of framing phrases (if any) consistent would allow programmers to quickly identify if a solution is being presented.

We distinguished between 11 different framing phrases over 24 hints, which can be seen in Table 2.3.

Framing Phrases	**Directions**	**Suggestions**	**Questions**	**Total**	**Percentage Use**
however, ...	1	0	0	1	5%
suggestion: ...	0	3	0	3	14%
hint: ...	0	2	0	2	10%
possible solution: ...	0	1	0	1	5%
possible fix: ...	0	**4**	0	4	19%
consider ...	0	1	0	1	5%
it could also mean that you need to ...	0	1	0	1	4%
maybe ... instead	0	1	0	1	4%
you may want to ...	0	2	0	2	5%
you can ...	0	1	0	1	5%
perhaps you ...	0	0	3	3	10%
did you ...	0	0	**4**	4	19%
Total	**1**	**16**	**7**	**24**	

Table 2.3: The distributions of the identified framing phrases among different hint types

From Table 2.3 we can see that there is no overlap between the framing phrases used for the different hint types. Additionally, we can observe that there are two framing phrases that were the most popular within certain hint types:

- "`possible fix: ...`", which makes up 31% of the framing phrases in the Suggestions hints; and
- "`did you ...`", which makes up 57% of the framing phrases in the Questions hints.

However, framing phrases are not always used. This is highly dependent on the category of hint, as is visualised in Figure 2.4.

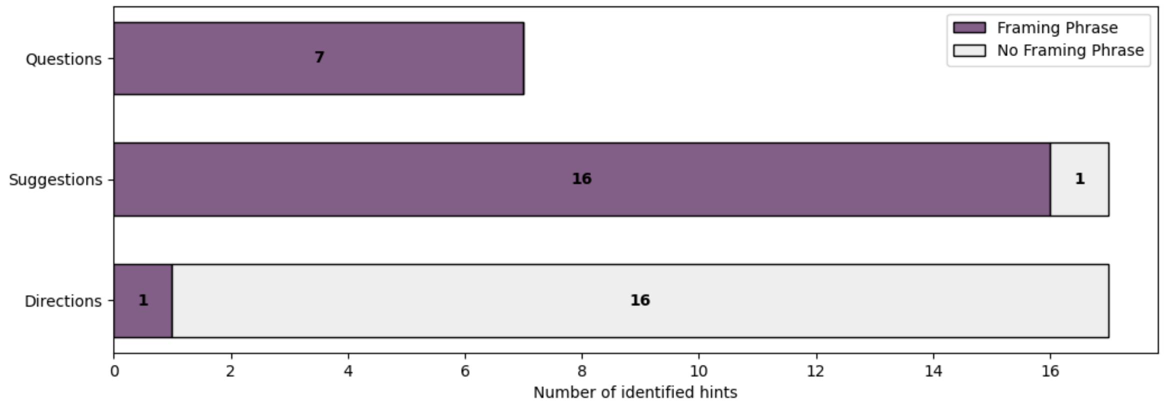


Figure 2.4: The distribution of hints found with and without framing phrases

We can see that framing phrases are almost always used for hints that are presented as Suggestions or Questions, indicating that *some* form of framing phrase has to be used. However, Directions hints tend to not have framing phrases, so future hints of this type should avoid them to be consistent with existing messages.

2.2.3 Structural Presentation

Another aspect of the hints that can be used to unify them is their structural presentation. Based on the identified hints, we split the structural presentations into the following categories:

- The hint is presented as a separate sentence from the error message.
- The hint is a direct continuation from the base error message, with no delimiting punctuation.
- The hint is within parentheses.

The distribution of these categories can be seen in Table 2.5.

Structural Presentation	**Directions**	**Suggestions**	**Questions**	**Total**	**Percentage Use**
New sentence	**10**	**10**	2	22	54%
Sentence continuation	2	1	0	3	7%
Parenthetical	5	6	**5**	16	39%
Total	**17**	**17**	**7**	**41**	

Table 2.5: The distributions of the identified structural presentations among different hint types

We can see that the structural presentation is dependent on the "type" of error message: Directions and Suggestions hints are mostly (59% of the time) presented as separate sentences to the base hint, whilst Questions hints are usually (71% of the time) parenthetical.

2.3 How to Proceed With this Information

The results show a lack of consistency between the existing hints regarding the information order, framing phrases used, and their structural presentation. As there is no clear winner for the most popular style of hint, we cannot give a definitive answer as to *which* format(s) new hints should use. However, we can provide the following (very) tentative advice for keeping a new hint consistent with prior hints:

- All new hints should follow the "Instruction → Purpose" order of information.
- A new hint of the Directions type should be a separate sentence with *no* framing phrases.
- A new hint of the Suggestions type should be a separate, capitalised sentence with "`Possible fix:`" as a framing preamble.
- A new hint of the Questions type should be within parentheses, and be enclosed in the "`did you ... ?`" framing phrase.

However, it should be noted that this advice is specifically for how to keep new hints *consistent* with existing hints. We make no statement as to *which* of these hint formats or types are best to use in any other measures. Furthermore, we strongly recommend that further work be done to unify the syntax patterns and structural presentations of the existing hints in Agda to reduce cognitive load on programmers.

2.4 Limitations

There are several factors to take into account when considering the representativeness of this investigation. These exist both in the identification of the hints, and the analysis of the findings.

Unfortunately, we cannot guarantee that all the hints that exist in the Agda codebase were identified. Despite trying to examine as many error messages as possible, we only looked at a subset of the code. Modules outside the `Error` and `Warning` modules that were investigated can potentially also have their own error messages (with accompanying hints) that were not identified. Furthermore, the identification was done manually, by reading over every error message. This allows for human error to influence the number of identified hits. As the sample size of the hints was very small, this could dramatically influence the results of the survey.

Furthermore, not everyone may agree with our definition of "hints" that were used in the identification process. One of the necessary criteria to be considered a "hint" was that it was secondary information. However, this meant that two similar error messages that arguably contain the same information are treated differently. For example, despite conveying similar information to the user, Listings 2.1 and 2.2 are categorised differently: Listing 2.1 is augmented with a hint, whilst Listing 2.2 is not.

$$\underbrace{\texttt{Universe Prop is disabled}}_{\text{error message}}\ \underbrace{\texttt{(use options --prop and --no-prop to enable/disable Prop)}}_{\text{hint}}$$

Listing 2.1: Error message content from `NeedOptionProp` in `Agda.TypeChecking.Errors`

$$\underbrace{\texttt{Option --rewriting needed to add and use rewrite rules}}_{\texttt{error message}}$$

Listing 2.2: Error message content from `NeedOptionRewriting` in `Agda.TypeChecking.Errors`

The small sample size of existing hints exacerbates these issues, and can reduce both the validity and reliability of the results. Additionally, this means that any future work on adding hints to error messages in Agda is going to significantly change the distributions of the identified features within the hints.

Finally, the results looked at the number of *distinct* hints that were implemented in the codebase, and were not influenced by how common they were. Unfortunately, identifying which hints were most commonly displayed was out of scope for this survey. This means that the distributions of hint formatting features that we obtained may not be representative of what a programmer actually experiences.

Chapter 3

Implementation

To be able to investigate the usability of EEMs enhanced with hints in Agda, it is necessary to have samples of such error messages.

All the "enhanced" error messages were implemented in a personal fork of the Agda project[1], with the goal of merging them into the Agda repository. The hints were added to existing error messages by modifying the `Agda.TypeChecking.Pretty.Warning` and `Agda.TypeChecking.Pretty.Call` modules. Whilst the `Agda.TypeChecking.Pretty.Warning` module provides "pretty" string representations of errors to programmers, `Agda.TypeChecking.Pretty.Call` is used to provide additional context to errors.

Three error messages were selected that would get augmented with hints:

Hint	Mistake Explanation	Error Type
Missing Space	Forgetting whitespace between variables or operators	`[NotInScope]`
Unicode Confusables	Using the wrong visually confusable Unicode character	`[NotInScope]`
Too Few Args	Not enough arguments passed to a function	`[UnequalTerms]`

These were error messages that were found to be confusing by novices taking the CSE3100 Functional Programming course[2] at TU Delft, a course which covers both Haskell and Agda. We also believed these error messages to give the best outcome for the user study without having to make too many changes to the Agda compiler. This made them good for investigating if further hint enhancements to error messages in Agda is worthwhile, and what direction such work should take.

The two `[NotInScope]` errors are raised during *scope checking*, wherein the scope checker verifies that all the variables, functions, and operators used are in scope. Meanwhile, the `[UnequalTerms]` error message is raised during the process of type-checking code, where the constraints of the types are verified and enforced.

This section will discuss the design and implementation details for the hints for each of the error messages. It will go over why each error message was selected, what we wanted the hint to look like, and what it looks like in the implementation. Furthermore, we provide a discussion of the shortcomings of our implementation, with recommendations for future work into implementing these hints.

3.1 General Design and Implementation Decisions

Before the design and implementation of each individual hint is discussed, we first explore a few general decisions that we made. In this subsection, we will first review how we used the results of Chapter 2 to inform the hint structure that we used. This will be followed up by a

[1] `https://github.com/mkhakimova42/agda`

[2] `https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=67585`

discussion of why we did not re-run the type-checker for each potential suggestion, even if it would have guaranteed that "incorrect" hints would not be presented.

3.1.1 Maintaining Consistency with Existing Hints

In Chapter 2 we identified that a significant portion (41%) of existing hints were presented as directions to the programmer. However, literature warns against this presentation, as we "cannot know the actual intent of the programmer" [3]. Hence, we decided to look at the other two options: presenting the hint as a suggestion (statement) or a question.

Although we identified significantly fewer hints structured as questions as opposed to suggestions, we also looked at the other hints that were presented in similar contexts to decide between these two presentations. From Appendix A, we can see that the most relevant existing hints are:

- `(did you forget space around the ':'?)`
- `(did you forget space around the '→'?)`
- `(did you mean` [**SUGGESTION**] `?)`
- `(did you supply too many arguments?)`

The first three were the only hints that we identified that accompany `[NotInScope]` errors, making them useful guidelines for our additional `[NotInScope]` hints. Furthermore, the last hint was a direct parallel to the `[UnequalTerms]` hint that we want to propose for *too few* arguments being supplied to a function. This overwhelming majority use of structuring the hints as `(did you ... ?)` questions led us to consider using the same format for our new hints. This allowed them to stay consistent with hints that programmers are used to seeing *in this context*.

Based on this information, we decided to structure all of our hints to have the parenthetical `did you ... ?` format. To stay consistent with other error messages of this format, the information order we used was just "Assumption". The rest of the error message was kept the same, as the goal was to investigate the effect of enhancing error messages with *hints*, and not enhanced error messages in general.

3.1.2 Not Re-Running the Type-Checker for Suggestions

Ideally, the hint would be presented only in situations when applying the fix suggested by the hint would "fix" the code[3]. However, given the scope of the research, and the limitations of the Agda compiler, we were not able to do this.

When there is only one error in the code, it would be possible to determine if the fix was useful (assuming that there was a way to have Agda type-check all of the generated code with the potential fixes): if the type checking succeeds, then the fix is useful, and should be proposed to the programmer.

Unfortunately, there may be situations when error that we are enhancing is not the only problem in the code, and fixing it makes the compiler display another error message. The previous strategy would then not work, as type-checking would still fail, even if the fix is correct. Thus, the hint would not get displayed in this situation. Furthermore, since the Agda compiler only knows one error[4] at a time, the number of error messages cannot be used as a heuristic to determine if the fix "improved" the situation either. Checking if the error that is encountered *changes* does not make for a good heuristic either, as that can also be a result of the applied fix *causing* the error.

[3]Although even this is arguable, as we cannot know the intentions of the programmer [3]

[4]Note that this does not apply to warnings unless the `-W error` or `--warning=error` option is used [34].

The Trade-Off

As a result of not being able to *accurately* verify the usefulness of all the hints with the type-checker, we had to consider the following trade-off between the following options:

- We do not re-run the type-checker on suggestions: the hint is displayed too frequently, even in cases where the suggestion does not fix the error.
- We re-run the type-checker on suggestions: the hint is displayed too infrequently, not appearing in cases where the suggestion would fix the error.

Not Re-Running the Type-Checker If the error message is too frequent and displays irrelevant suggestions, the programmer might be led down the wrong path to solving the error, which may lead to significantly more confusion if the programmer trusts the hint to be correct. This effect may be stronger in novice programmers, who may not trust their ability to write Agda code. This in turn can make programmers ignore the hints in the long run, making the enhancement pointless.

Re-Running the Type-Checker If the hint is not presented frequently enough, the programmer may not attempt the actions that are usually suggested to them by hints. This can happen even when they are the "correct" fix, as they are used to the hint giving that guidance. The sporadic presentation of the hints from the point of view of the programmer can exacerbate this, as before the error is resolved they cannot tell that there is another error in the code. This can make programmers spend much longer trying to fix the code.

Furthermore, this would require the type-checker to be re-run for each potential suggestion. Running the Agda type-checker is slow[5], especially on large projects that use external libraries. However, even when using heuristics to determine if a suggestion is plausible, there can still be several proposed solutions. Re-running the type-checker for of these solutions can dramatically increase the amount of time a programmer has to wait to type-check their code, decreasing the usability of the overall system. Since novice programmers commonly repeat errors, and are good indication that the student is struggling [35, 2], this increased compilation time might disproportionately affect novice programmers[6].

Conclusion In the end, we chose to display the hints too frequently, as we hypothesised that the novices would be able to recover from the incorrect hint without too much difficulty, and the experimental setup would allow us to test this claim. This means that all of the EEMs can potentially display irrelevant suggestions which can lead programmers "down the wrong path", as Marceau *et al.* warned [4].

Although there may be a work-around for both the efficiency and effectiveness of an implementation that does re-run the type-checker, implementing and testing this was out of scope for this project. We leave to future work.

3.2 Missing Space

When a programmer forgets mandatory spaces around mixfix operators, they can encounter a `[NotInScope]` error thrown by the type-checker[7]. In prior research, Juhošová *et al.* identified this to be a common mistake made by novice Agda programmers [25].

[5]While type-checking usually takes several seconds for larger files, this can go up to the order of minutes in extreme cases.

[6]Although it should be noted that novices are also less likely to work on larger files.

[7]It should be noted that `[NotInScope]` is not the only type of error that can arise from this mistake, with other possibilities like `[NoParseForLHS]`, or `[CannotApply]`. However, for the purposes of investigating if this enhancement is useful, we will be focusing on the `[NotInScope]` instance of the hint. The same applies to the other error messages discussed in this section

Consider the following function from the `Data.Product.Base` module in the Agda standard library [36]:

```
swap : A × B → B × A
swap (x , y) = (y , x)
```

Notice the spaces around the `,`. These are necessary for the comma to get interpreted as the defined mixfix operator. Now consider that the programmer forgets a space, as is custom with using commas in regular text and most programming languages:

```
swap : A × B → B × A
swap (x , y) = (y, x)
```

In popular programming languages like Java and Python, one would not expect omitting space prior to a comma to affect the code's output. However, attempting to load an Agda file with this code will result in the following error message:

```
/home/m/Documents/error/examples/swap.agda:8,17-19: error: [NotInScope]
Not in scope:
  y, at /home/m/Documents/error/examples/swap.agda:8,17-19
when scope checking y,
```

Without the space before the comma, `y,` gets interpreted as one name, instead of a variable `y` followed by an operator `,`. This is due to a fundamental design decision in Agda, wherein names are allowed to use almost all Unicode characters, excluding whitespace and the special characters `.;{}()@"` [37]. This feature is used frequently in Agda code, including the Agda standard library. Take as an example the following code from `Algebra.Properties.Group` [38]:

```
x•y⁻¹≈ε⇒x≈y : ∀ x y → (x • y ⁻¹) ≈ ε → x ≈ y
x•y⁻¹≈ε⇒x≈y x y x•y⁻¹≈ε = begin
  x         ≈⟨ inverseˡ-unique x (y ⁻¹) x•y⁻¹≈ε ⟩
  y ⁻¹ ⁻¹   ≈⟨ ⁻¹-involutive y ⟩
  y         ∎
```

As we can see, the names use non-standard Unicode characters very liberally. This leads to the unusually strong requirement to place whitespace between all names and operators, which might surprise novice Agda programmers. Looking at the example above, `x•y⁻¹≈ε` is a variable of type `(x • y ⁻¹) ≈ ε`, and the two cannot be used interchangeably. This type of naming convention is common in the Agda community [39].

However, this behaviour can be interpreted as being inconsistent. Whitespace is not necessary around reserved special characters, as they cannot be used within a valid name. This means that `(x)` is equivalent to `( x )`, whilst `[x]` and `[ x ]` have different meanings (given a user or library-defined operator `[_]`). This can be confusing to novices who are not used to such behaviour.

Whilst this behaviour is unusual amongst popular programming languages, Agda is not the only language that imposes strict whitespace usage. For example, MetaMath[8] is even stricter than Agda, requiring space even around normal brackets `()`. If one were to attempt to prove that `2 + 2 = 4`, the goal would be the following:

```
|- ( 2 + 2 ) = 4
```

However, assume that the programmer omits spaces as such:

[8] `https://us.metamath.org/`

```
|- (2 + 2) = 4
```

Upon encountering this proof step, the metamath-lamp[9] proof assistant interprets `(2` as one identifier, and displays the following error:

```
The symbol '(2' is not declared.
```

However, such languages are not mainstream, and Agda novices are unlikely to have encountered them before tackling Agda. Therefore, it is likely that they will not be familiar with such restrictions.

3.2.1 Designing the Hint

Before implementing the EEM in Agda, it first had to be designed. Taking the previous example:

```
swap : A × B → B × A
swap (x , y) = (y, x)
```

Recall that the error message displayed to the programmer was the following:

```
/home/m/Documents/error/examples/swap.agda:8,17-19: error: [NotInScope]
Not in scope:
  y, at /home/m/Documents/error/examples/swap.agda:8,17-19
when scope checking y,
```

As decided in Section 3.1.1, the hint was to be structured as `(did you mean ... ?)`, leaving the base error message unchanged. The enhanced error message thus added line 4 to the original error message, resulting in the following being displayed to the programmer:

```
/home/m/Documents/error/examples/swap.agda:8.17-19: error: [NotInScope]
Not in scope:
  y, at /home/m/Documents/error/examples/swap.agda:8.17-19
    (did you forget whitespace in  'y ,'?)
when scope checking y,
```

If there are multiple viable hints, they get appended to the hint as such:

```
(did you forget whitespace in
       'x xx' or
       'xx x'?)
```

3.2.2 Implementing the EEM

We have implemented the hint in the `Agda.TypeChecking.Pretty.Warning` module, which can be seen in `https://github.com/mkhakimova42/agda/pull/3`. The changes include a new function, `didYouMeanInfix` which is called by `prettyWarning` when the warning type is `NotInScopeW`. This results in it being appended to the standard `[NotInScope]` error, similar to how the other `(did you ... ?)` hints are displayed.

Our implementation only uses one heuristic. When the (unqualified) name that is not in scope can be broken down *completely* into (unqualified) names that *are* in scope[10], all possible "breaks" are displayed as a hint. This is done according to the design described in Section 3.2.1.

[9] `https://lamp-guide.metamath.org/`

[10] This is sometimes known as a Word Break problem [40].

3.2.3 Limitations of the Implementation

The main limitation of our implementation was the frequency at which the hint got presented. Ideally, the hint would be presented only in situations when the `[NotInScope]` error was caused by missing whitespace, and adding the space as dictated would resolve it. However, as discussed in Section 3.1.2, we were not able to re-run the type-checker to guarantee that the error would improve the situation. This meant that all possible solutions that satisfy scope checking are presented, even if they do not satisfy type checking.

3.3 Unicode Confusables

Novice programmers can often get tripped up by both Agda's support for Unicode characters in the names of functions and variables, and the ease of entering these characters in text editors with interactive support for Agda. Since the set of Unicode characters contains many homoglyphs (i.e. characters that are visually similar or identical to each other[11]), it can often be difficult to identify if an incorrect character was accidentally entered. This is a relatively common complaint from both students at TU Delft learning Agda, and users on the internet [41]. Similar complaints have also been bought up in other languages that offer similar Unicode support [42].

As an example, look at the two following equals signs, which can look identical to each other in certain fonts:

=	＝
003D	FF1D
EQUALS SIGN	FULLWIDTH EQUALS SIGN

Both of these symbols can be easily typed in Emacs agda-mode: the equals sign by typing `=`, and the fullwidth equals sign by typing `\=`. However, they are interpreted as different characters, and cannot be used interchangeably. It is therefore very easy to type one in place of the other accidentally, leading to an error in the code.

Such characters are very common, and widely used in Agda. A common complaint with both the Agda built-ins[12] and standard library is that they both use ∷ instead of `::` for list cons [41, 43]. With certain fonts, these can look indistinguishable from each other, especially if the programmer is not expecting it. This can be confusing to Agda novices coming from Scala or Rocq which use `::` for the same operation [44, 45]. With the prevalence of both Unicode confusables and the usage of Unicode in important Agda libraries, this is a frequent hurdle for novices (and programmers) to overcome.

However, there are other ways of accidentally inputting the wrong character. It is easy to accidentally enter a cyrillic с[13] if the programmer has both Russian JCUKEN and English QWERTY keyboard layouts installed, as the two are located on the same key. In Rust, developers found that hundreds of projects used a fullwidth exclamation mark (U+FF01) in place of a normal exclamation mark (U+0021) using a CJK input method [31]. If the programmer accidentally switches inputs, they will almost certainly notice that incorrect characters are being entered and delete them. However, if the incorrect input started on a homoglyph, it is likely that the programmer will keep that symbol as it looks "correct". Since Agda supports Unicode in variable and function names, this will not be highlighted as an issue until type-checking is run again, and the name is found to be out of scope.

[11] In the context of Unicode, these are often referred to as Unicode confusables.

[12] `https://agda.readthedocs.io/en/v2.7.0.1/language/built-ins.html`

[13] `с` (U+0441) as opposed to `c` (U+0063).

It is no surprise then that efforts have already been made to provide useful information for this scenario. In Emacs agda-mode, typing `M-x describe-char` or `C-u C-x =` gives information about the character the cursor is positioned over [46]. For example, for the fullwidth equals ＝ we get the following:

```
            character: ＝ (displayed as ＝) (codepoint 65309, #o177435, #xff1d)
              charset: unicode (Unicode (ISO10646))
code point in charset: 0xFF1D
               script: cjk-misc
               syntax: w    which means: word
             category: .:Base, c:Chinese, h:Korean, j:Japanese, |:line breakable
             to input: type "\F=" or "\=" or "\eq" with Agda input method
          buffer code: #xEF #xBC #x9D
            file code: #xEF #xBC #x9D (encoded by coding system utf-8-unix)
              display: by this font (glyph code):
    ftcrhb:-PfEd-AR PL UMing TW MBE-light-normal-normal-*-27-*-*-*-*-0-iso10646-1
    ↪  (#x57AB)

Character code properties: customize what to show
  name: FULLWIDTH EQUALS SIGN
  general-category: Sm (Symbol, Math)
  decomposition: (wide 61) (wide '=')
```

However, this is not a perfect solution. The programmer needs to both know that this command exists, and think to check that for the two characters that have been confused. If the wrong confusable character was used within a larger string, this can be challenging to identify. Furthermore, this option is only available in Emacs. Thus, programmers who use a different coding environment (e.g. VS Code, or Vim) will not have easy access to this information[14].

The best way to provide this information such that the code can be fixed easily is thus through a hint within the error message. Rust is a famous example that already provides a hint if it thinks that the programmer incorrectly inputted a Unicode confusable. Implementing such a hint has also been considered several times in Agda [49, 43], although the issue has been put into the icebox[15] each time.

3.3.1 Designing the Hint

Currently, typing a Unicode confusable in place of the expected character causes the standard `[NotInScope]` error. For example, consider the following case where the latin `c` is used on the left hand side, but the cyrillic `с` was typed on the right:

```
add3 : Nat → Nat → Nat → Nat
add3 a b c = a + b + с
```

Upon loading the file, the following error message is displayed:

```
/home/m/Documents/error/examples/cyrillic-c.agda:4,22-23: error: [NotInScope]
Not in scope:
  с at
/home/m/Documents/error/examples/cyrillic-c.agda:4,22-23
when scope checking с
```

[14]There do exist VS Code extensions that can provide the code point of a Unicode character [47, 48], but no references are made to them in the Agda documentation. A novice programmer would have to actively seek them out to obtain this information.

[15]This milestone is described as containing “Issues there are no plans to fix for upcoming releases” in the Agda repository.

The goal was thus to add a hint to this error message that tells the programmer that they used a different `c` to the one in scope, with the relevant information as to how to fix it.

The Inspiration

We took the Rust error message as inspiration for this hint. Consider the following case, where the programmer typed a Greek question mark instead of a semicolon:

```
fn main() {
  println!("Hello, world!");
}
```

Compilation of this code fails with the following error:

```
error: unknown start of token: \u{37e}
 --> greek_question_mark.rs:2:30
  |
2 |     println!("Hello, world!");
  |                             ^
  |
help: Unicode character ';' (Greek Question Mark) looks like ';'
↪  (Semicolon), but it is not
  |
2 -     println!("Hello, world!");
2 +     println!("Hello, world!");
  |

error: aborting due to 1 previous error
```

Here we can see that the following information is provided to the user:

- the reason the compilation failed: the wrong Unicode character was used,
- the location of the wrong Unicode character,
- information about the wrong Unicode character: the hex code and the name, and
- a suggestion for the correct Unicode character, with its name.

However, this error message is only presented for parse errors, and not when a name is not in scope due to a confusable character. Nevertheless, there is a warning that displays upon the usage of identifiers that look alike:

```
fn main() {
  let α = 5;
  c = 𝛼
}
```

This code compiles with two errors and three warnings, of which the following warning is most relevant:

```
warning: found both `α` and `𝛼` as identifiers, which look alike
 --> alpha_warning_with_err.rs:3:9
  |
2 |     let α = 5;
  |         - other identifier used here
3 |     c = 𝛼
  |         ^ this identifier can be confused with `α`
  |
  = note: `#[warn(confusable_idents)]` on by default
```

This hint is always displayed[16], even if the code compiles without issue. However, when it is displayed in conjunction with a "cannot find value in this scope", pointing at the same location, it can potentially be inferred that the error was due to the mixed script confusable.

This is not desirable for Agda because using confusable characters is baked into the Agda programming community. For example, Unicode considers `×` (U+00D7) to be confusable with `x` (U+0078) [50], but the Agda standard library frequently uses both[17]. Therefore, constantly presenting such a warning can often be irrelevant, increasing cognitive load and reducing the usability of the error and warning messages.

Our Hint

In designing the hint, we prioritised the following information to display to the user:

- the location of the confusable character(s) within the name,
- what the confusable characters are (with their identifiers), and
- how to input both the correct and incorrect characters with Emacs agda-mode.

This was a situation where colour-coding the error message would have been very useful, for example to indicate what characters are "wrong" and which are "correct". However, this was not done as many supported text editors (like VS Code) did not support the colour-coding of error messages that was present in Emacs. Since this also applied to the WebLab editor that was used in the user study to investigate the *impact* of the enhanced error messages (refer to Chapter 4), the study would not have been able to draw any conclusions about the usefulness of this change.

With these considerations in mind, the following hint was implemented:

```
(did you accidentally use a confusable character?
   You typed:    [TYPED STRING]
   In scope:     [IN SCOPE STRING]
   (diff)        [DIFF]

         Character                       Character name      agda-mode input     Emacs input
   ---->  [TYPED CHAR] (0x[HEX CODE])      [CHARACTER NAME]    [INPUT OR BLANK]    C-x 8 RET [HEX CODE]
      vs  [IN SCOPE CHAR] (0x[HEX CODE])   [CHARACTER NAME]    [INPUT OR BLANK]    C-x 8 RET [HEX CODE]

   OR
   You typed: ...                                                                                    )
```

Therefore, given the following code:

```
id : Set → Set
id bαp = baρ
```

[16]Assuming that `#[warn(confusable_idents)]` is on, which it is by default.

[17]For example, both are used in the same file in `Data.Nat.Properties`: `https://agda.github.io/agda-stdlib/v2.2/Data.Nat.Properties.html`

The `(did you accidentally use a confusable character?)` hint is appended to the standard list of hints:

```
/home/m/Documents/error/examples/alpha.agda:2.10-13: error: [NotInScope]
Not in scope:
  b𝕒ρ at /home/m/Documents/error/examples/alpha.agda:2.10-13
    (did you accidentally use a confusable character?
       You typed:    b𝕒ρ
       In scope:     bαp
       (diff)        -^^

              Character        Character name                      agda-mode input  Emacs input
       ---->  '𝕒' (0x1d552)    MATHEMATICAL DOUBLE-STRUCK SMALL A    \ba              C-x 8 RET 1d552
          vs  'α' (0x3b1)      GREEK SMALL LETTER ALPHA              \Ga              C-x 8 RET 3b1

       ---->  'ρ' (0x3c1)      GREEK SMALL LETTER RHO                \Gr              C-x 8 RET 3c1
          vs  'p' (0x70)       LATIN SMALL LETTER P                                   C-x 8 RET 70
                                                                                                   )
when scope checking b𝕒ρ
```

3.3.2 Implementing the Hint

The hint was implemented in the `Agda.TypeChecking.Pretty.Warning` by adding a new function, `didYouMeanConfusableUnicode` which is called by `prettyWarning` when the warning type is `NotInScopeW`. The code can be seen in `https://github.com/mkhakimova42/agda/pull/2`.

The hint is displayed whenever there is at least one (unqualified) name in scope where all of the characters (in order) are either exactly the same as each other or are confusable with each other (with at least one character that is not equal between the two strings [18]). To determine if characters are confusable, we used the `text-icu` package[19] that uses the `confusablesWholeScript.txt`[20] file for spoof-checking [51].

The Emacs input keybindings are displayed following the standard input pattern: `C-x 8 RET [codepoint hexadecimal]` [52]. However, the process for obtaining the agda-mode keybindings is more complicated. The agda-mode Unicode input keybindings are currently defined in the `agda-input.el` file[21]. However, we found an initiative by Chonavel to move them into a JSON file [53], in order to make the keybindings system-wide [54]. Although this feature is now in the "later" milestone[22], we used the JSON file that came from it due to ease of parsing.

3.3.3 Limitations of the Implementation

Unfortunately, there were several limitations of our implementation of the Unicode Confusables hint. This subsection will discuss several of these limitations: those related to the displayed keybindings, which strings are considered confusable, hint frequency, and display issues arising from non-standard Unicode characters. From this, we also give recommendations for future work into *this* particular error message enhancement.

Reliability of the Keybindings File

It should be noted that we did not use a primary source for our agda-mode keybindings. Instead, we used the JSON file created by Chonavel [53] in 2024. This means that the keybindings file may not represent the actual keybindings, and this discrepancy may grow as

[18]This check is necessary because the existing `(did you mean ... ?)` hints are computed on unqualified names.

[19]`https://hackage.haskell.org/package/text-icu-0.8.0.5`

[20]`http://unicode.org/Public/security/latest/confusablesWholeScript.txt`

[21]`https://github.com/agda/agda/blob/master/src/data/emacs-mode/agda-input.el`

[22]This milestone contains "features that the development team might eventually implement, but not for the next release" [55].

`agda-input.el` gets updated. In fact, we have already identified and rectified a couple of these errors in the pull request, highlighting this issue. In future work, we recommend that the original `agda-input.el` file be used for providing the keybindings in the hint (unless they have been moved to a different location).

Is < Confusable with ⟨?

The package used to determine if two characters are confusable used the `confusablesWholeScript.txt`[23] file for spoof-checking [51]. However, there are many characters that are important to Agda developers that are not covered by `confusablesWholeScript.txt`, such as:

- `::` (two U+003A characters) and ∷ (U+2237), as identified by Michael Nahas [43],
- `<` (U+003C) with ⟨ (U+27E8), as identified by Frederik Hanghøj Iversen [49], and
- `=` (U+003D) with ＝ (U+FF1D), the latter of which can be accidentally inserted with agda-mode input by typing `\=`.

However, in future work on this hint it should be possible to remedy these issues, as a different `confusablesWholeScript.txt` can be used, tailored to the needs of Agda users. We also recommend that a survey of the commonly confused characters in Agda be conducted, such that a proper `confusablesWholeScript.txt` can be created for this purpose.

Confusability is Only Checked Against Names in Scope

The current implementation of the hint only checks the name that is not in scope against names that are in scope. However, there is also a chance that the programmer used a confusable character instead of a *reserved name*[24]. Since these are stored in a different place to the names in scope, the hint is not shown when such an error is made (e.g. when 𝜆 (U+1D706) is typed instead of λ (U+03BB)). In future work on this hint, we recommend that reserved names are also checked against.

Hint Frequency

Ideally, the hint would be presented when substituting the typed name for either a name that is in scope (with the differences between the two being only the confusable characters) would fix the error. However, as discussed in Section 3.1.2, we cannot determine if a change fixed that particular error.

This results in the hint appearing too frequently, as there is a chance that the error will not be resolved correctly upon applying the fix. However, the chances of this happening are slim, and likely indicate that the programmer needs to pick variable and function names that are visually distinguishable from each other.

The limitations detailed prior also contributed to the error message *not* appearing frequently *enough*. This can lead the programmer to discount the possibility of confusable character when error does not display hint, as they are used to the hint getting displayed when it is relevant. However, further work on refining the error messages, as discussed in those sections, should be enough to remedy these issues.

Misaligned Display

Unfortunately, the diff arrows can get misaligned when upon the presence of characters that do not have the standard width (e.g. fullwidth characters). For example, if we take this extreme example:

[23] `http://unicode.org/Public/security/latest/confusablesWholeScript.txt`
[24] `https://agda.readthedocs.io/en/latest/language/lexical-structure.html`

```
id : Set → Set
id a=bＡc3ａdefc = a=bAc3adefc
```

The diff (using JuliaMono[25]) looks like this, with the relevant characters underlined for clarity:

```
You typed:    a=bac3adefc
In scope:     a=bＡc3ａdefc
(diff)        ---^--^----
```

We can see the misalignment getting progressively worse with each consecutive fullwidth character. This significantly hinders the readability of the hint in cases where non-standard with characters are used.

Whilst it could be possible to follow Rust's example and display 2 upticks for full-width characters, that would not fully resolve the problem. The extent of the misalignment is highly font-dependent — in some mono fonts fullwidth characters take up the same space as two normal characters, whilst in others they take slightly less space. Since we cannot predict what font the programmer will use, this is difficult to account for. Furthermore, this would not fix the issue that the two strings get misaligned with each other as well. The best way to remedy this would be to colour code the confusable characters (as was done above), but with the current state of the supported text editors, this would leave many programmers with incomplete information.

3.4 Too Few Args

Another frequent mistake or typo programmers can make (in any language) is accidentally supplying the wrong number of arguments to a function [57, 58]. However, while there is already a hint for when too many arguments are passed to a function, the same does not apply to when there are too few. Instead, the following error message is frequently presented:

```
addThree : Nat → Nat → Nat → Nat
addThree a b c = a + b + c

num : Nat
num = addThree 1 2
```

```
/home/m/Documents/error/examples/not-enough-args.agda:7.7-19: error:
↪  [UnequalTerms]
Nat → Nat !=< Nat
when checking that the inferred type of an application
  Nat → Nat
matches the expected type
  Nat
```

Unfortunately, this error message can be confusing to novice programmers, especially when they are not comfortable parsing Agda types.

3.4.1 Designing the Hint

Before creating a design for the hint, we looked at other similar hints for inspiration. This subsection covers these sources of inspiration, before continuing on to the design of the error message.

[25]This is the font recommended by the *Programming Language Foundations in Agda* book [56].

The Inspiration

Recall that Agda already has a hint when too *many* arguments are passed to a function: `(did you supply too many arguments?)`. This was used as the base for the design of this hint.

Haskell and Rust also already incorporate a hint for when too few arguments are supplied to a function. An example of such an error message in Haskell can be seen below:

```
main :: IO ()
main = do
  putStr $ show $ addThree 1 2

addThree :: Int -> Int -> Int -> Int
addThree a b c = a + b + c
```

```
addThree.hs:3:12: error: [GHC-39999]
    • No instance for ‘Show (Int -> Int)’ arising from a use of ‘show’
        (maybe you haven't applied a function to enough arguments?)
    • In the first argument of ‘($)’, namely ‘show’
      In the second argument of ‘($)’, namely ‘show $ addThree 1 2’
      In a stmt of a 'do' block: putStr $ show $ addThree 1 2
  |
3 |   putStr $ show $ addThree 1 2
  |            ^^^^
```

We looked at Haskell's error message for inspiration as Agda novices are likely to have prior experience with Haskell, and might already be familiar with the hint. Additionally, as the function types are displayed similarly, a clear parallel could be drawn between the two.

Here, we can see that the error message has the hint `(maybe you haven't applied a function to enough arguments?)`. However, it does not present information about what function is missing the arguments, or what the arguments should be.

Rust's error message for the same mistake is more informative:

```
fn main() {
  println!("{}", add_three(1, 2));
}

fn add_three(a: u32, b: u32, c:u32) -> u32 {
  return a + b + c
}
```

```
error[E0061]: this function takes 3 arguments but 2 arguments were supplied
 --> add_three.rs:2:17
  |
2 |     println!("{}", add_three(1, 2));
  |                    ^^^^^^^^^------ argument #3 of type `u32` is missing
  |
note: function defined here
 --> add_three.rs:5:4
  |
5 | fn add_three(a: u32, b: u32, c:u32) -> u32 {
  |    ^^^^^^^^^                 -----
help: provide the argument
  |
2 -     println!("{}", add_three(1, 2));
2 +     println!("{}", add_three(1, 2, /* u32 */));
  |

error: aborting due to 1 previous error
```

Rust was used as inspiration due to its reputation for having useful hints. This error message provides the following information:

- the function that is missing argument(s): its name and location of definition,
- the position of the missing argument(s), and
- the required type(s) of the missing argument(s).

The Hint

When designing the hint, it was important to consider the practical limitations given by the scope of the project and the Agda compiler.

Unfortunately, it was not possible to include all the desirable features that we identified in our inspiration. It was infeasible to provide the location of the function definition within the scope of the project. Furthermore, given the current structure of the Agda compiler, we were not able to find a way to consistently give correct information about the location or type of the missing argument(s). This meant that the resulting error message was closest to the one implemented in Haskell and Agda, only with the name of the function that is missing the arguments present:

```
addThree : Nat → Nat → Nat → Nat
addThree a b c = a + b + c

num : Nat
num = addThree 1 2
```

```
/home/m/Documents/error/examples/not-enough-args.agda:7.7-19: error:
↪  [UnequalTerms]
Nat → Nat !=< Nat
when checking that the inferred type of an application
  Nat → Nat
matches the expected type
  Nat
(did you supply too few arguments to addThree ?)
```

3.4.2 Implementing the EEM

The error message is shown when the `CheckTargetType`[26] function in the `Agda.TypeChecking.Pretty.Call` module is called and the inferred type of the application is a function type. We have implemented it modifying `CheckTargetType` (see `https://github.com/mkhakimova42/agda/pull/1`).

3.4.3 Limitations of the Implementation

There are several limitations of our hint implementation that should be considered. This subsection discusses shortcomings in the amount of detail that is displayed in the hint, as well as in the frequency of the hints. We also provide advice for future work on this hint.

The Amount of Detail in the Hint

Unfortunately, the hint that we implemented does not provide as much detail as we would have liked, as it is missing

- the location where the function that is missing argument(s) is defined,
- the position of the missing argument(s), and
- the type(s) of the expected argument(s),

In future work, we recommend that this information be presented in this hint.

Hint Frequency

While it would be ideal to have the error message present only when fixing the error would make it type-check, this was not possible, as discussed in Section 3.1.2. Unfortunately, this resulted in the hint showing up too frequently, even when the proposed fix is not applicable.

However, there are also cases where the hint does not show up when needed. One such case is when the programmer forgets to supply the function with any arguments, since `CheckTargetType` does not get called when *no* arguments were supplied to the function:

```
addThree : Nat → Nat → Nat → Nat
addThree a b c = a + b + c

num : Nat
num = addThree
```

```
/home/m/Documents/error/examples/not-enough-args.agda:7.7-15: error:
↪  [UnequalTerms]
Nat → Nat → Nat → Nat !=< Nat
when checking that the expression addThree has type Nat
```

While it might be useful to ask the programmer if they intended to supply `addThree` with arguments in this case, the hint does not appear.

Additionally, sometimes the inferred type of the expression can be simplified in a function type (and therefore would warrant the hint), but the hint is not displayed as this simplification cannot be done from `CheckTargetType`. This leads to situations like the following:

[26]This is the function that provides the additional context "`when checking that the inferred type of an application ... matches the expected type ...`".

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
  begin
    (suc m + n) + p
  ≡⟨ cong suc (+-assoc m n) ⟩
    suc m + (n + p)
  ∎
```

```
/home/m/Documents/error/examples/args-not-simplified.agda:12,16-27: error:
↪  [UnequalTerms]
(p₁ : ℕ) → m + n + p₁ ≡ m + (n + p₁) !=< m + n + p ≡ m + (n + p)
when checking that the expression +-assoc m n has type
m + n + p ≡ m + (n + p)
```

In future work, we strongly recommend that better heuristics be used to display the hint.

Chapter 4

Experimental Setup

In order to answer the research questions detailed in the introduction, we conducted a user study. In it, we asked participants to resolve errors in existing code with the help of the original and enhanced error messages, and then rate the helpfulness of the error messages they received. This chapter will detail the structure of the user study: how the participants were recruited, how the survey was deployed, and what the survey consisted of. Additionally, Section 4.4 describes the pilot study used to validate the structure and deployment method of the user study. The user study was approved by the TU Delft Human Research Ethics committee.

4.1 Recruitment

The participants of the user study were students of the 2024/2025 edition of the CSE3100 Functional Programming course at TU Delft[1]. This is a 10-week Bachelor course that is taught in English, where students received 8 lectures on Haskell, and then 4 on Agda.

The course content for the Agda lectures can be seen in table Table 4.1.

Lecture 1	**Lecture 2**	**Lecture 3**	**Lecture 4**
Agda basics	Dependent Types	The Curry-Howard Correspondence	Equational Reasoning in Agda

Table 4.1: The lecture structure in CSE3100: Functional Programming

These participants were recruited halfway through Lecture 4. An announcement was made then about the user study, and they were given 10 minutes of lecture time, with the option to continue into the 15-minute mid-lecture break, to complete it on their own laptops. Additionally, an announcement was posted on the course Brightspace page — an online learning environment used by TU Delft [59]. This announcement can be seen in Appendix B. The survey remained open for 11 days, between the 31st of March and the 10th of April.

Before gaining access to the questions of the user study, the participants had to agree to the informed consent form, which can be seen in Appendix C.

4.2 Deploying the Survey

The user study was held on the CS3100: Functional Programming WebLab page[2], which was already used for the prior coding assignments in the course. It was chosen as students

[1] `https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=67585`

[2] `https://eip.pages.ewi.tudelft.nl/eip-website/weblab.html`

were already familiar with the environment, with many courses that students would have previously taken in the Computer Science & Engineering Bachelor programme using WebLab. This resulted in a low risk of confusion about the WebLab interface influencing the results.

Additionally, WebLab allows for the setup of different programming environments (through the use of Docker images) for different assignments. This meant that we could easily have different participants receiving different versions of Agda for the same question, allowing us to test the influence of the enhanced error messages in comparison to the original ones.

4.3 The Experiment

To evaluate the usefulness of the enhanced error messages, participants of the study were given 10 programming questions to solve. In the programming questions, the participants were given existing code with one error in it[3]., and they were expected to fix the error using the error message(s) that they received from the compiler. After each programming question they were asked to rate the error message that they received on a Likert scale.

This section will provide an overview of the programming questions that were created, the Likert scale used, as well as general design decisions for the user study.

4.3.1 Programming Questions

The goal of the programming questions was to answer our research questions and sub-questions:

- **RQ1** *Which kinds of hints or solutions provided in Agda's error messages are valued by novices?*
- **RQ2** *To what extent does providing novices with solutions or hints in Agda's error messages improve their ability to fix errors?*
    - **RQ2.1** *How does enhancing Agda's error messages with hints influence the probability that novices are able to resolve the error?*
    - **RQ2.2** *How does enhancing Agda's error messages with hints affect the time novices spend fixing the error?*

The programming questions featured pre-written code that contained one error that would cause the enhanced error message to display, which was partially inspired by Becker, Goslin, and Glanville [11]. This was meant to simulate being in the middle of coding, or debugging someone else's code. The goal of the exercise was to try to compile the code, read the error message that gets displayed, and then fix the error, re-compiling the code as needed until the error was resolved, and the code compiled. The timestamps of the submissions were then used to determine the time spent on each question, and the compilation status was used to determine the success rate of each question.

Since there were cases where the enhanced error messages can provide both correct and incorrect suggestions, the user study investigated the effect of both. To do this, we created five groups of questions for the user study, which can be seen in Table 4.2.

To allow each student to see both the enhanced and original error message for each type of error message, two different programming questions were created for each group, resulting in 10 questions.

These questions were created by taking inspiration from the existing WebLab exercises, to keep consistent with both the content and difficulty of the course. This meant that we did not import the Agda standard library, despite its prevalence in Agda code. Instead, the exercises come with student-visible code that has the needed functionality. The syntax of this

[3]A `[NotInScope]` or `[UnequalTerms]` error that evokes one of the enhanced error messages discussed in Chapter 3.

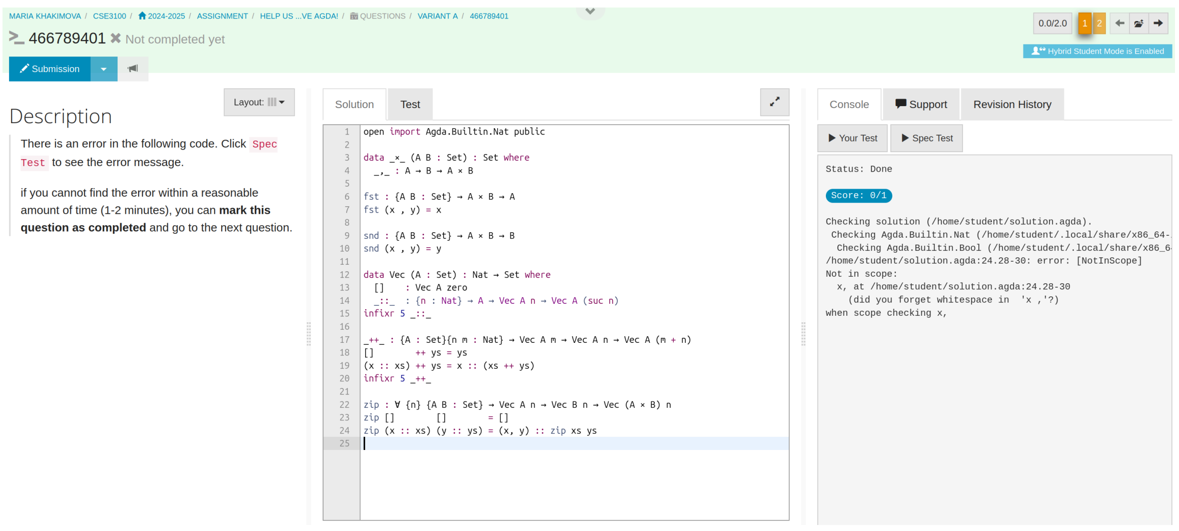
Question Group	Hint Suggestion	Actual Solution
Missing Space: **Correct**	Add whitespace in given place	Add whitespace in given place
Missing Space: **Incorrect**	Add whitespace in given place	Fix a different typo in same location
Unicode Confusables: **Correct**[5]	Use a different Unicode character in given place	Use a different Unicode character in given place
Too Few Args: **Correct**	Supply one more argument to given function	Supply one more argument to given function
Too Few Args: **Incorrect**	Supply one more argument to given function	Use a different function / variable in place of the given function

Table 4.2: The five groups of questions used in the user study

"library" code is also often different from that in the Agda standard library, with less Unicode (excluding printable ASCII) characters, as WebLab does not support Emacs or agda-mode Unicode insertion methods. This is consistent with how the assignments in the CSE3100: Functional programming course were structured.

Additionally, some exercises were adapted from code from the Agda standard library[6], the Iowa Agda library[7], and the *Programming Language Foundations in Agda book*[8].

An example of a programming question in the WebLab environment can be seen in Figure 4.3. The full list of all the programming questions, with their accompanying original and enhanced error messages, as well as the expected "corrected" code, are made publicly available in the accompanying repository [60].



MARIA KHAKIMOVA / CSE3100 / 2024-2025 / ASSIGNMENT / HELP US ...VE AGDA! / QUESTIONS / VARIANT A / 466789401
466789401 Not completed yet
Submission
0.0/2.0
1 2
Hybrid Student Mode is Enabled
Layout:
Solution
Test
Console
Support
Revision History
Your Test
Spec Test

Description

There is an error in the following code. Click `Spec Test` to see the error message.

if you cannot find the error within a reasonable amount of time (1-2 minutes), you can **mark this question as completed** and go to the next question.

```
open import Agda.Builtin.Nat public

data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

fst : {A B : Set} → A × B → A
fst (x , y) = x

snd : {A B : Set} → A × B → B
snd (x , y) = y

data Vec (A : Set) : Nat → Set where
  []    : Vec A zero
  _::_  : {n : Nat} → A → Vec A n → Vec A (suc n)
infixr 5 _::_

_++_ : {A : Set}{n m : Nat} → Vec A m → Vec A n → Vec A (m + n)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
infixr 5 _++_

zip : ∀ {n} {A B : Set} → Vec A n → Vec B n → Vec (A × B) n
zip []        []        = []
zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

```
Status: Done

Score: 0/1

Checking solution (/home/student/solution.agda).
 Checking Agda.Builtin.Nat (/home/student/.local/share/x86_64-
  Checking Agda.Builtin.Bool (/home/student/.local/share/x86_6
/home/student/solution.agda:24.28-30: error: [NotInScope]
Not in scope:
  x, at /home/student/solution.agda:24.28-30
    (did you forget whitespace in  'x ,'?)
when scope checking x,
```

Figure 4.3: The WebLab interface on a programming question

The same instructions as can be seen on the left of Figure 4.3 were given for each question. It was requested that students "mark [the] question as completed" upon abandon in order to log the amount of time that was spent on the question before abandoning. If this was not done for a submission, its timing data would have to be discarded.

[6]https://agda.github.io/agda-stdlib/
[7]https://github.com/cedille/ial
[8]https://plfa.github.io/

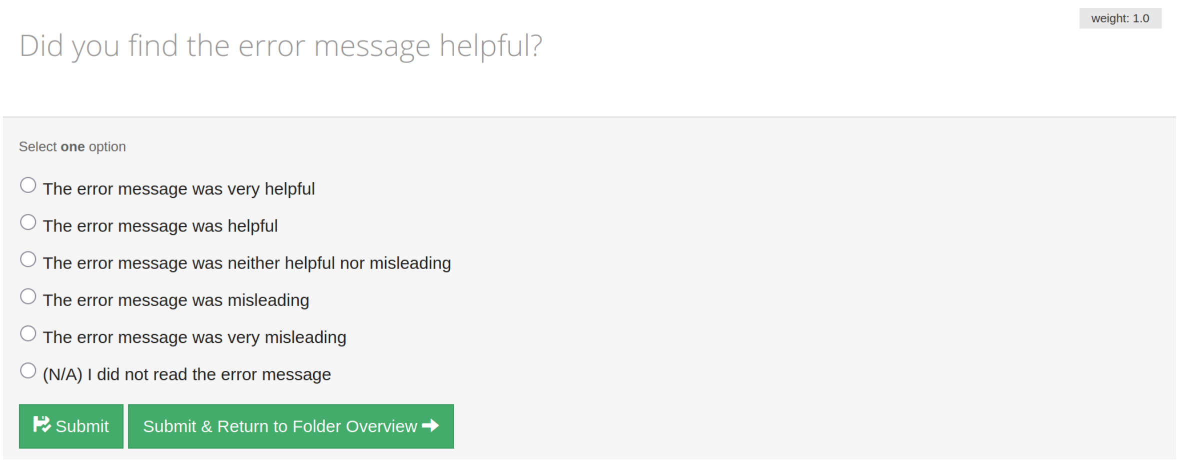
4.3.2 Opinions on the Error "Helpfulness"

After each programming question, the participants were asked to rate the "helpfulness" of the error message they just received. This was done to answer **RQ1**: *Which kinds of hints or solutions provided in Agda's error messages are valued by novices?*

A five-point bipolar Likert scale was chosen to collect this information. The two poles were that the error message was "very helpful" and "very misleading", allowing us to see the extent to which incorrect hints are detrimental, and if correct hints are ever considered detrimental. The odd number ensured that there was a midpoint "neutral" option, allowing participants to say that they did not find the error message helpful or misleading.

A 6th "(N/A): Did not read the error message" option was added to ensure that people who did not read the error message did not skew the results. Additionally, it served to distinguish the neutral "neither helpful nor misleading" from simply not reading the question, minimising confusion about the meaning of that option.

The full Likert scale as seen in WebLab can thus be seen in Figure 4.4.



weight: 1.0

Did you find the error message helpful?

Select **one** option

- ○ The error message was very helpful
- ○ The error message was helpful
- ○ The error message was neither helpful nor misleading
- ○ The error message was misleading
- ○ The error message was very misleading
- ○ (N/A) I did not read the error message

Figure 4.4: The Likert scale, as displayed in WebLab

4.3.3 Experiment Design

A between-subjects design was chosen for the experiment, with two variants of the survey being created. This meant that participants could not use their prior experience with the code to fix the mistake instead of the error message, reducing learning effects. This also came with the benefit of reducing the time taken for the survey, leading to less fatigue and potentially increasing the participation rate. However, this resulted in a smaller sample size, making the results less generalisable.

Two variants of the survey were created, which were then randomly assigned to the participants using WebLab's "Variants" feature. An overview of these variants can be seen in table Table 4.5.

For each category of question, each student received one exercise with the original error message, and one with the enhanced error message. This was done to diminish the influence of interpersonal differences on the results per error message category.

The participant-visible names of each question were also replaced by 9-digit random numbers, to reduce the chances of participants recognising a pattern in the name and what the error is / which type of error message they will receive.

Since participants were not forced to complete the survey, there was a high chance that many participants would not answer all the questions. This was exacerbated by the length of the survey, which could lead to fatigue. To prevent this from disproportionately affecting

Question	Hint Correctness	Displayed Error Message	
		Variant A	Variant B
`Missing Space (A)`	Correct	Enhanced	Original
`Missing Space (B)`	Correct	Original	Enhanced
`Missing Space (C)`	Incorrect	Enhanced	Original
`Missing Space (D)`	Incorrect	Original	Enhanced
`Unicode Confusables (A)`	Correct	Enhanced	Original
`Unicode Confusables (B)`	Correct	Original	Enhanced
`Too Few Args (A)`	Correct	Enhanced	Original
`Too Few Args (B)`	Correct	Original	Enhanced
`Too Few Args (C)`	Incorrect	Enhanced	Original
`Too Few Args (D)`	Incorrect	Original	Enhanced

Table 4.5: The ten programming questions used in the user study

certain questions, all the questions were randomly shuffled for each participant. This gave each participant a different question order, resulting in most questions having similar participation rates. This also served to diminish the influence of fatigue and learning effects on the results.

4.4 Pilot Study

Before conducting the experiment, a pilot experiment was performed on 4 people with varying levels of experience with Agda. Three of the participants had higher levels of experience with Agda than the students, and one was an Agda novice. Half of the participants received Variant A, and the other half Variant B. The purpose of this pilot was to:

- identify if there were any issues with the WebLab setup,
- proof-check for mistakes,
- ensure the difficulty level was at the right level, and
- get an estimate of how long the full survey tok.

The difficulty was judged to be appropriate by the participants, but the length was unexpectedly long. On average, the more experienced participants took 16 minutes, whilst the Agda novice took 35 minutes.

This justified the initial decision of shuffling the exercises, as it would even out the distribution of responses for each question even if participants drop the user survey due to it taking too much time. Furthermore, we also added a line to the description of each question urging participants to continue to the next question if they could not complete it "within a reasonable amount of time (1-2 minutes)" to increase the number of participants who completed the full survey. This can be seen on the left side of figure Figure 4.3.

Chapter 5

Results

The following chapter will present the aggregated and processed results of the user study. First, we will discuss the test that we will use to test for the statistical significance of our results in Section 5.1. We then present the participation rates of our survey in Section 5.2. Section 5.3 will present the data that was collected to answer **RQ1**, which will be followed by Section 5.4 detailing the results relevant to **RQ2**. The unprocessed (anonymised) results can be viewed in the accompanying repository [60].

5.1 Statistical Significance: The Mann-Whitney U Test

To test for significance in the differences between the enhanced and original error messages, Mann-Whitney U (MWU) tests were used. This is a statistical test to "compare two independent groups that do not require large normally distributed samples" [61]. As we did not have large sample sizes, and our data was not normally distributed, this was an appropriate choice.

In a Mann-Whitney U test, there are two hypotheses:

- the null hypothesis H_0, which states that there is no difference in the distributions of the two groups, and
- the alternative hypothesis H_1, which is against H_0.

The alternative hypothesis depends on if a two- or one-tailed test is chosen. In a two-tailed test, H_1 stipulates that the distributions of the two groups differ, although no comment is made on which direction that difference is in. This would have been useful if we wanted to test if there was *any* difference between the enhanced error messages (EEMs) and original error messages (OEMs).

However, a single-tailed test has an alternative hypothesis that suggests the *direction* of the difference between the two group (either positive or negative). In our case, this would have allowed us to test if EEMs are better or worse than OEMs (but not both at the same time). Since knowing the direction of the difference was valuable to evaluating the usefulness of the EEMs, this was the more appropriate choice.

We thus decided to perform a single-tailed test, where we varied the direction of H_1 depending on whether we were analysing the results for "correct" or "incorrect" EEMs. This allowed us to test the following:

- Were the "Correct" EEMs were *more* useful than the OEMs?
- Were the "Incorrect" EEMs were *less* useful than the OEMs?

5.1.1 Reporting the p Values

To determine if the evidence for the alternate hypothesis was statistically significant, all analyses used a significance level of $p < 0.05$, as was proposed by Fisher [62]. The tables with the reported p values have the rows for which the difference was statistically significant highlighted to aid readability.

We also reported the actual p values as equalities, unless the p value was less than 0.001, in which case we reported it as $p < 0.001$. This is in accordance with the common guidance on p value reporting [63].

5.2 Programming Question Response Rate

To put the results into context, it can be useful to discuss the response rate of the survey. Out of the 70 participants who started the survey by opening a submission, the random assignment of variants led to similar response rates between the variants. This can be seen in Figure 5.1.

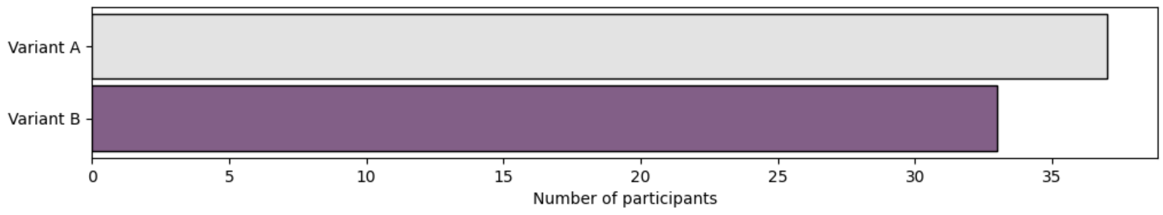


Figure 5.1: The number of participants who received the survey variants

However, we cannot guarantee that all the participants gave legitimate responses to every question. Many submissions were abandoned: the participants opened the submission, but made no changes to the code within WebLab, giving no timing data. This distribution of abandoned submissions can be seen in Figure 5.2.

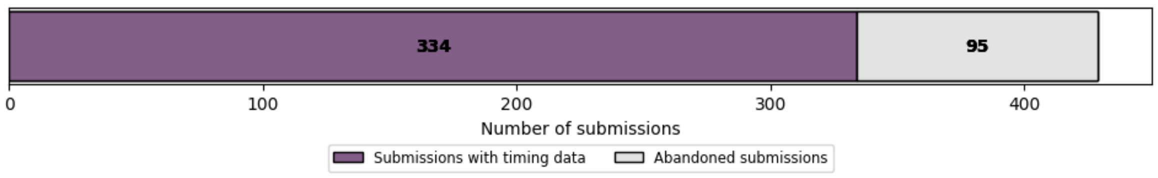


Figure 5.2: The total number of programming question submissions, those with timing data and those without

As we can see in Figure 5.3, the number of responses per question varies a lot. This is both when looking at all the attempts, and just the ones with some timing data.

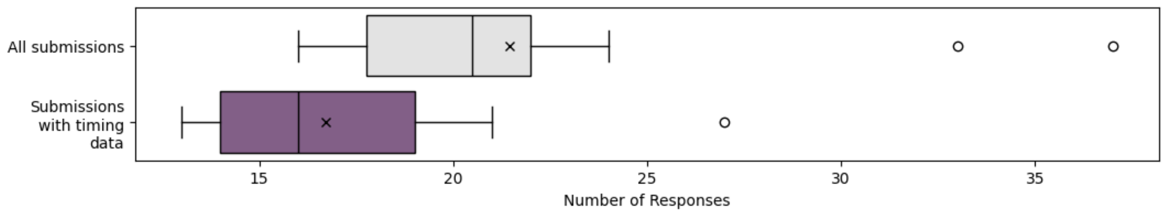


Figure 5.3: The distribution of the number of programming question submissions per question

5.3 Subjective Opinions of the Error Messages

This section will present the results of how helpful the novices found the EEMs. This information is necessary to answer **RQ1**. We used the following alternative hypotheses (H_1) for the Mann-Whitney U tests:

- When a “correct” hint is presented, enhanced error messages are rated as more *helpful* than the corresponding original error messages.
- When a “incorrect” hint is presented, enhanced error messages are rated as more *misleading* than the corresponding original error messages.

We first look at the results from a general perspective, identifying trends between the helpfulness of the “correct” and “incorrect” hints when compared to the OEM in Section 5.3.1. Then, we provide a more detailed overview of the results for the “correct” hints in Section 5.3.2, and the “incorrect” hints in Section 5.3.3.

5.3.1 General Trends in Novice Perception

A visualisation of the aggregated results for all the submissions can be seen in Figure 5.4. Furthermore, Table 5.5 presents the results of the single-tailed Mann-Whitney U test on this data.

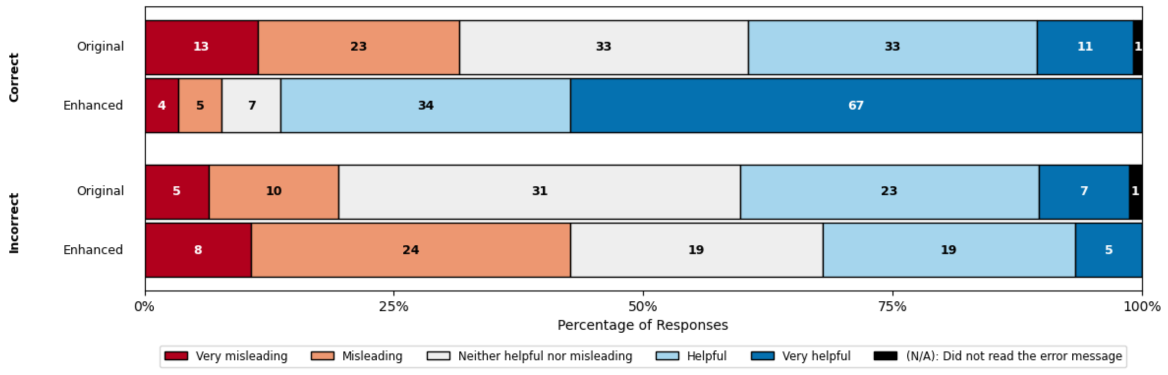


Figure 5.4: Overview of novices’ perceptions of the helpfulness of the error messages

EEM Hint Type	U Statistic	p Value	Statistical Significance ($p < 0.05$)
Correct	2595.5	<0.001	✓
Incorrect	3405.5	0.016	✓

Table 5.5: Aggregated results of the single-tailed MWU tests (perceived helpfulness)

From Figure 5.4 and Table 5.5 we can see that we have strong evidence to reject *both* null hypotheses:

- The EEMs with “Correct” hints are more helpful than their corresponding OEM.
- The EEMs with “Incorrect” are rated more misleading than their corresponding OEMs.

5.3.2 Per-Question Results for “Correct” Hints

Recall that the hypotheses used for the Mann-Whitney U tests in this subsection were the following:

- **Null hypothesis** (H_0)**:** There is no difference between the percieved helpfulness of the EEMs and the OEMs.
- **Alternative hypothesis** (H_1)**:** When a “correct” hint is presented, EEMs are rated as more *helpful* than the corresponding OEMs.

The main goal of this part of the investigation was the general perception of the EEMs. However, we also believed that it could be beneficial to investigate how the students who could *not* resolve the error rated the "helpfulness" of the error message[1]. Therefore, after discussing the trends in the general population, this section will also look at these "unsuccessful" submissions.

All Submissions

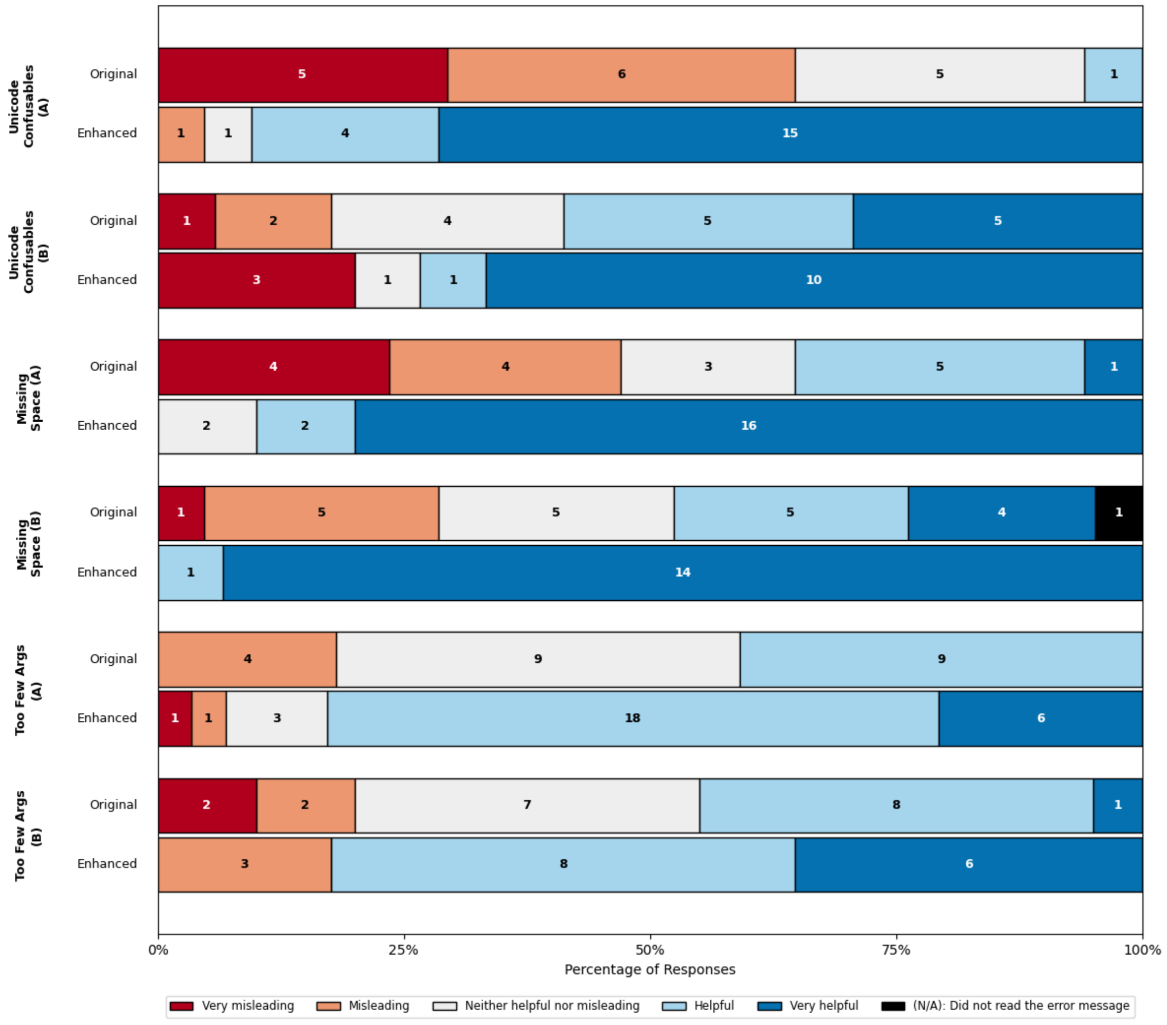


Figure 5.6: Perceptions of the error message helpfulness: all submissions ("Correct" hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Unicode Confusables (A)`	14.5	<0.001	✓
`Unicode Confusables (B)`	94	0.093	×
`Missing Space (A)`	30	<0.001	✓
`Missing Space (B)`	34.5	<0.001	✓
`Too Few Args (A)`	163.5	0.001	✓
`Too Few Args (B)`	94	0.008	✓

Table 5.7: Single-tailed MWU results: are "correct" hints more helpful (all submissions)?

[1]This includes the people who "abandoned" the question without resubmitting, and participants who changed and submitted their code at least once.

From Figure 5.6 we can identify a general trend of the “correct” EEMs being rated as more helpful than their OEM counterparts. Furthermore, the results in Table 5.7 indicate that we have sufficient evidence to reject the null hypothesis for all the programming exercises apart from `Unicode Confusables (B)`.

Unsuccessful Submissions

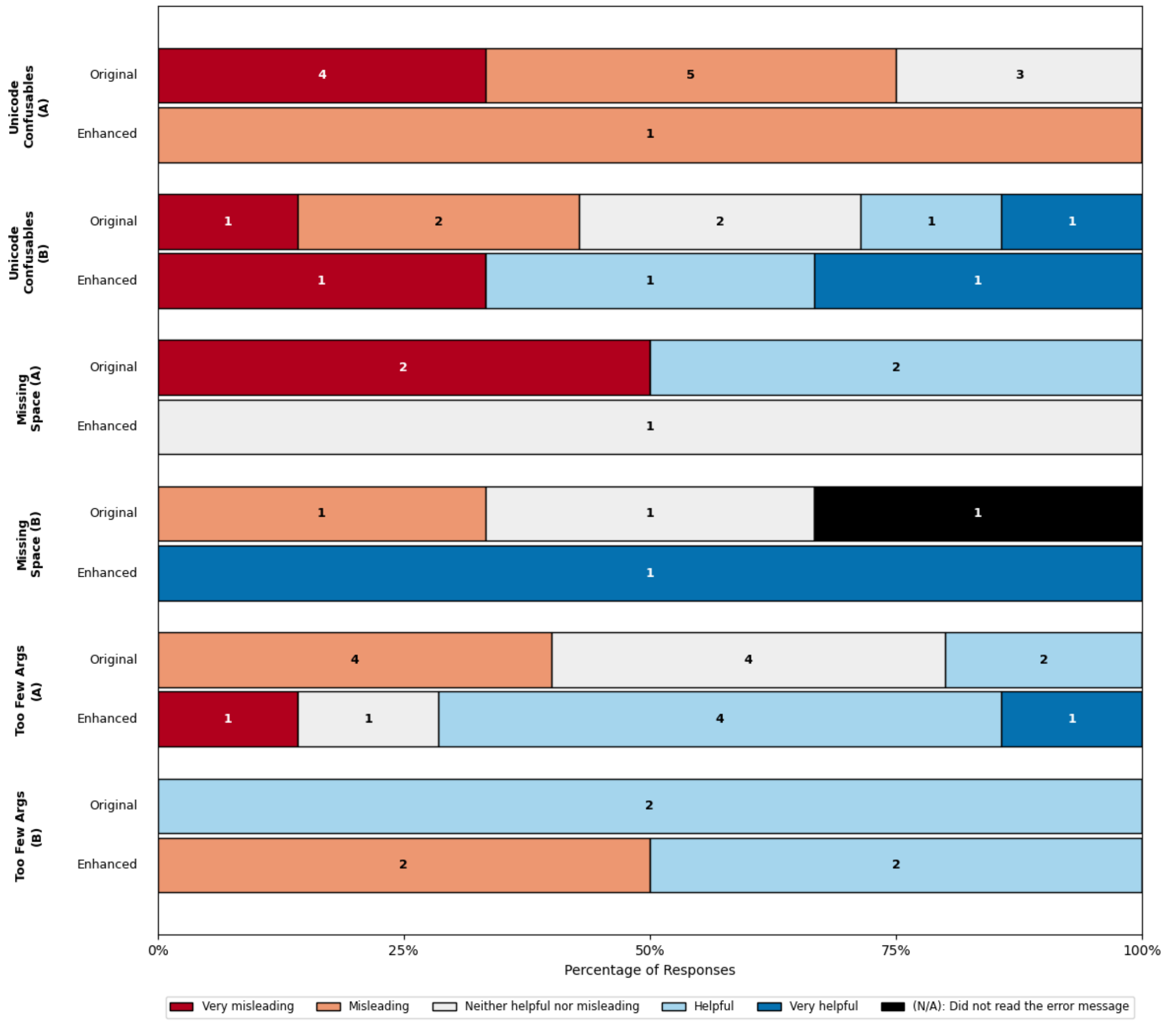


Figure 5.8: Perceptions of the error message helpfulness: unsuccessful submissions (“Correct” hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Unicode Confusables (A)`	5.5	0.500	×
`Unicode Confusables (B)`	8.5	0.364	×
`Missing Space (A)`	2	0.645	×
`Missing Space (B)`	0	0.333	×
`Too Few Args (A)`	18	0.047	✓
`Too Few Args (B)`	6	0.919	×

Table 5.9: Single-tailed MWU results: are “correct” hints more helpful (unsuccessful submissions)?

From Figure 5.8, we can see a different distribution of the data. Now, only three EEMs are rated as more helpful than the OEM. From Table 5.9, we can conclude that this difference is only statistically significant for `Too Few Args (A)`.

However, it should be noted that the sample size for these submissions were mostly very small, making it difficult to draw valid conclusions from this data.

5.3.3 Per-Question Results for “Incorrect” Hints

The hypotheses used for the “incorrect” hints were the following:

- **Null hypothesis** (H_0)**:** There is no difference between the perceived helpfulness of the EEMs and the OEMs.
- **Alternative hypothesis** (H_1)**:** When a “incorrect” hint is presented, EEMs are rated as more *misleading* than the corresponding OEMs.

Similar to what was done in Section 5.3.2, we will provide the results for all the submissions, as well as the subset of unsuccessful submissions.

All Submissions

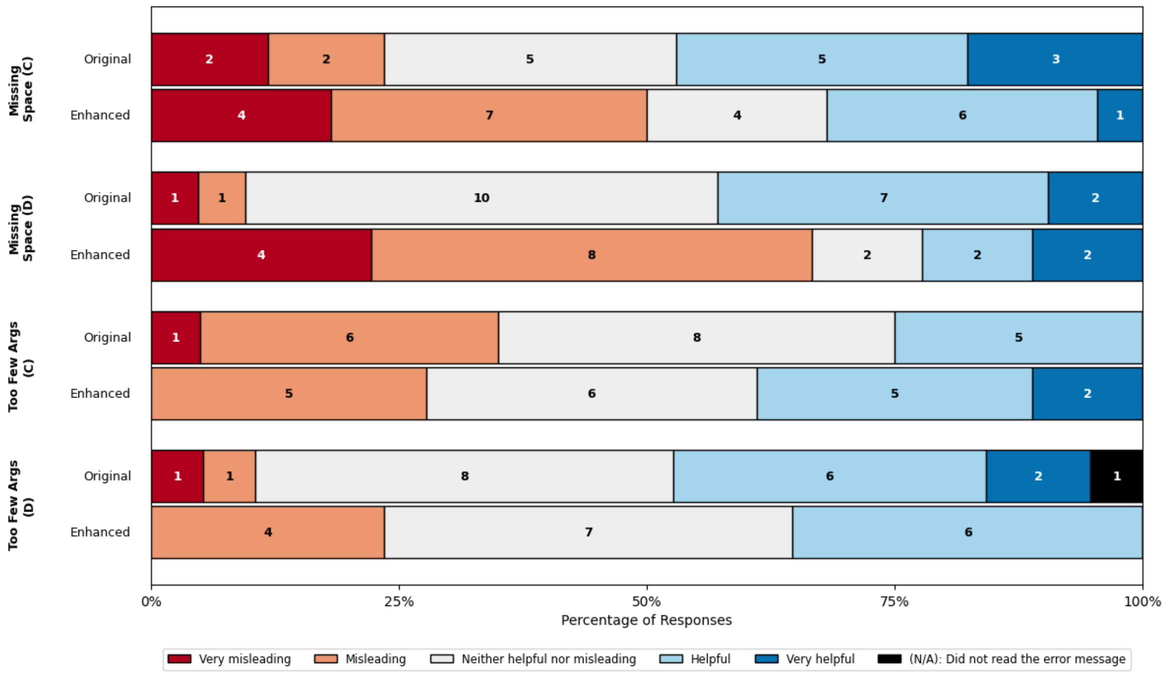


Figure 5.10: Perceptions of the error message helpfulness: all submissions (“Incorrect” hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Missing Space (C)`	238.5	0.069	×
`Missing Space (D)`	279	0.005	✓
`Too Few Args (C)`	146.5	0.852	×
`Too Few Args (D)`	180	0.175	×

Table 5.11: Single-tailed MWU results: are “incorrect” hints more misleading (all submissions)?

From Figure 5.10 we can see a general trend of EEMs with incorrect hints being rated as less “helpful” and more “misleading”. The only exception to this is `Too Few Args (C)`, which was still rated as more helpful than the OEM.

Given the results in Table 5.11, we could only accept the alternate hypothesis H_1 for one question: `Missing Space (D)`. For the other questions, there is *no* evidence suggesting that the "incorrect" EEMs are perceived as more misleading than the OEMs by Agda novices.

Unsuccessful submissions

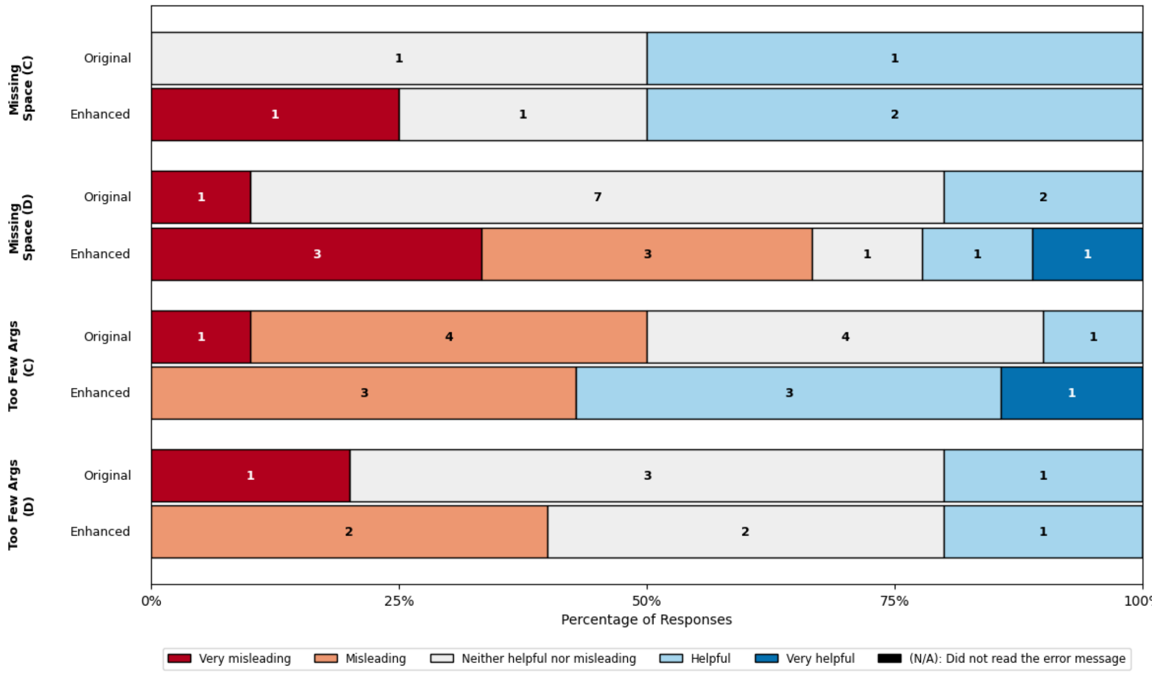


Figure 5.12: Perceptions of the error message helpfulness: unsuccessful submissions ("Incorrect" hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Missing Space (C)`	4.5	0.500	×
`Missing Space (D)`	62	0.079	✓
`Too Few Args (C)`	22.5	0.909	×
`Too Few Args (D)`	13.5	0.455	×

Table 5.13: Single-tailed MWU results: are "incorrect" hints more misleading (unsuccessful submissions)?

In Figure 5.12, we can see that the "incorrect" EEMs typically received a similar number of "(very) helpful" votes. However, more participants also rated them as "(very) misleading". The exception to this is again `Too Few Args (C)`, which participants found more helpful.

When comparing the statistical significance of the results from just the unsuccessful submissions in Table 5.13 to all the submissions in Table 5.11, we can see that our conclusions are the same: we can only reject the null hypothesis for `Missing Space (D)`.

5.4 Novices' Performance

To answer **RQ2**, as well as the sub-questions **RQ2.1** and **RQ2.2**, we used two objective measures for novice performance:

- the proportion of participants that were able to successfully locate and resolve the error (the success rate), and

- the time spent by participants to successfully resolve the error.

This section will thus discuss the data collected for both of these measures, with the results of the accompanying Mann-Whitney U tests.

5.4.1 Success Rate

The success rate is defined as the number of successful attempts opposed to non-successful attempts. This includes both cases where participants "abandoned" the question without making a submission, and those for which there is evidence of re-submissions.

The following alternate hypotheses are used for the one-tailed Mann-Whitney U analyses in this section:

- Participants who receive an "incorrect" enhanced error message have a *lower* success rate than participants who received the original Agda error message.
- Participants who receive an "correct" enhanced error message have a *higher* success rate than participants who received the original Agda error message.

In this subsection, we will discuss both the general trends in the success rates, before discussing the differences in the question success rates for the "correct" and "incorrect" EEMs.

General Trends in Success Rate

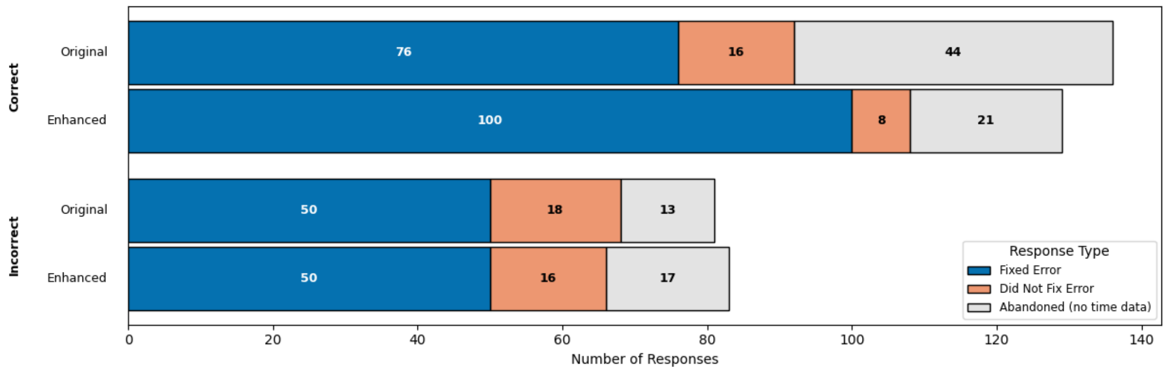


Figure 5.14: Aggregated number of responses per question

EEM Hint Type	U Statistic	p Value	Statistical Significance ($p < 0.05$)
Correct	10670	<0.001	✓
Incorrect	3311.5	0.424	×

Table 5.15: Aggregated results of the single-tailed MWU tests (success rate)

In general, there was no visible change in success rates for the "incorrect" hints. This gives no evidence in favour of the alternate hypothesis for "incorrect" hints. However, the increase in success rates for the "correct" hints was statistically significant, giving very strong evidence in favour of the "correct" EEM alternate hypothesis.

Per-Question Results for "Correct" Hints

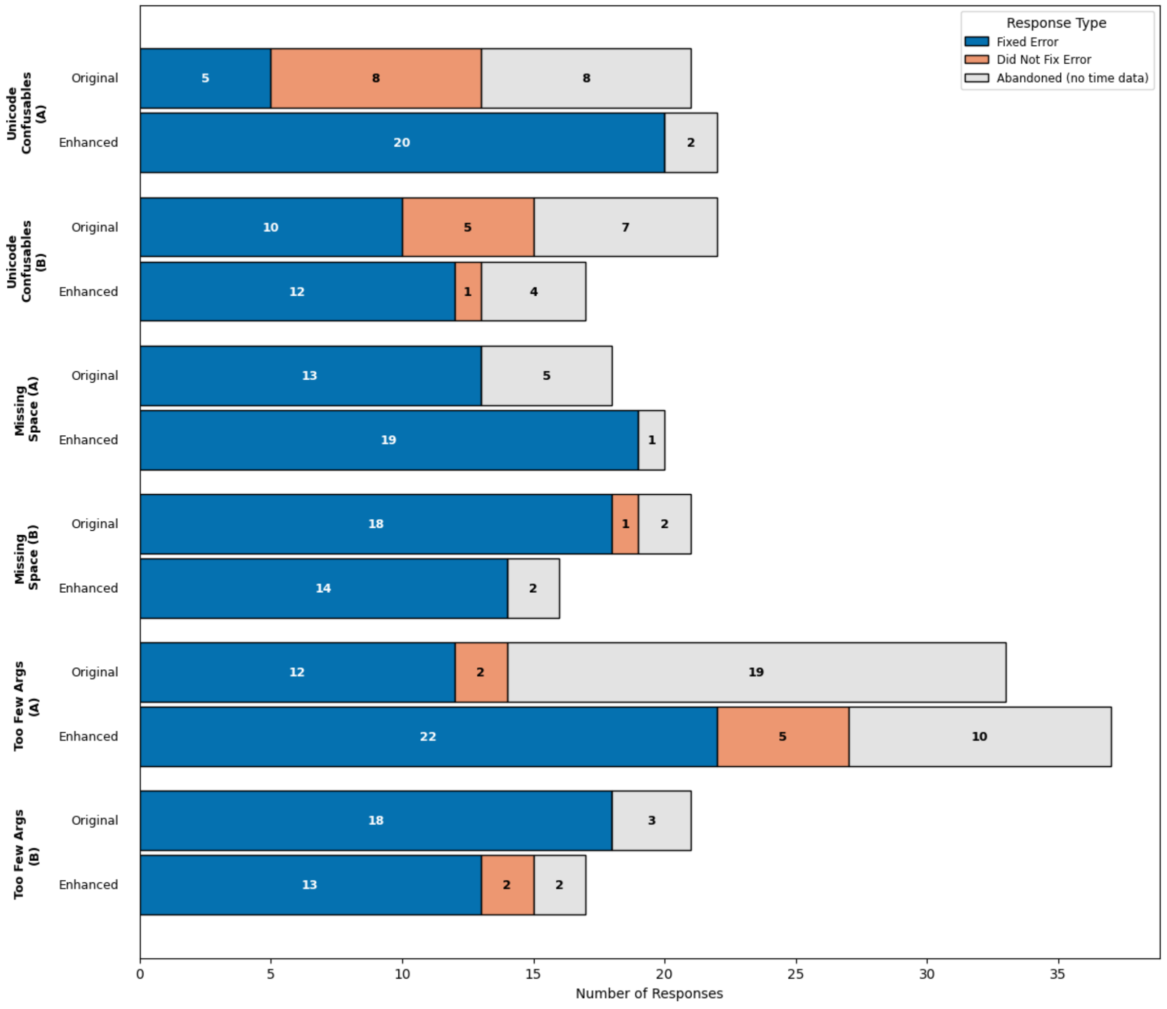


Figure 5.16: Number of responses per question ("Correct" hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Unicode Confusables (A)`	386	<0.001	✓
`Unicode Confusables (B)`	234	0.063	×
`Missing Space (A)`	221	0.030	✓
`Missing Space (B)`	171	0.449	×
`Too Few Args (A)`	751.5	0.028	✓
`Too Few Args (B)`	162	0.771	×

Table 5.17: Single-tailed MWU results: do "correct" hints increase the success rate?

Recall that the hypotheses were the following:

- **Null hypothesis** (H_0)**:** There is no difference between the success rates of participants receiving the EEMs and the OEMs.
- **Alternative hypothesis** (H_1)**:** Participants who receive an "correct" EEM have a *higher* success rate than participants who received the OEM.

From Figure 5.16, we can see that students who received EEMs had higher success rates for every question apart from `Too Few Args (B)`. However, when considering the MWU results in Table 5.17 we only have evidence supporting our H_1 for half of the questions: one from each category of hints.

Per-Question Results for “Incorrect” Hints

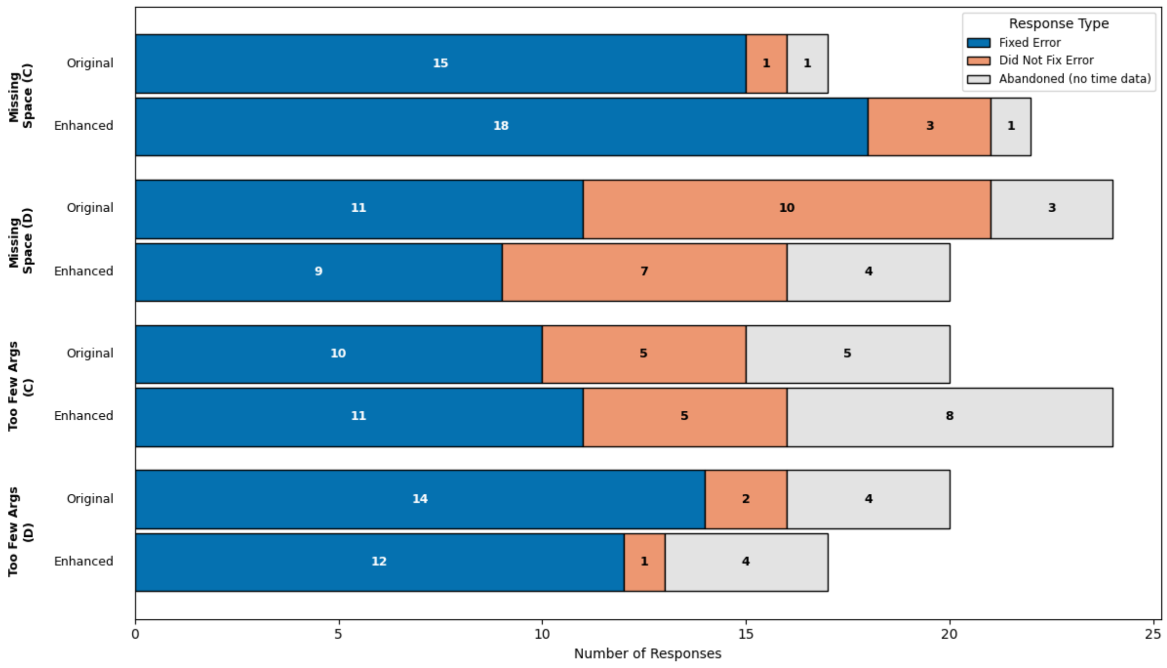


Figure 5.18: Number of responses per question (“Incorrect” hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Missing Space (C)`	175	0.301	×
`Missing Space (D)`	238	0.484	×
`Too Few Args (C)`	230	0.398	×
`Too Few Args (D)`	171	0.523	×

Table 5.19: Single-tailed MWU results: do “incorrect” hints decrease the success rate?

The hypothesis for the “incorrect” EEMs were that:

- **Null hypothesis** (H_0)**:** There is no difference between the success rates of participants receiving the EEMs and the OEMs.
- **Alternative hypothesis** (H_1)**:** Participants who receive an “incorrect” EEM have a *lower* success rate than participants who received the OEM.

In Figure 5.18, we can observe minor decreases in the percentage of successful submissions for three of the four questions (with a slight increase for `Too Few Args (D)`). However, the Mann-Whitney U results in Table 5.19 indicate that none of them were statistically significant. Thus, we maintain the null hypothesis for all the questions.

5.4.2 Timing

Before analysing the timing data, it was first necessary to clean the data to prevent inadvertent skews in the results. Therefore, this section will first discuss how the data was cleaned. We then present per-question results for the “correct” and “incorrect” hints, as due to differences in sample sizes and the ranges of times taken per question, we cannot provide reliable aggregated data.

When presenting the mean and median times in tables, we highlight the columns in which the difference between EEM and OEM timings are in the *expected* direction[2]. This is the direction corresponding to the alternate hypothesis.

Data Cleaning

Since the user study was open for 11 days, and no strict order to the questions was enforced, participants were free to stop a question at any point and return to it later. Whilst this likely increased participation rates per question and helped give more useful data for the success rates, this occasionally resulted in that data saying that a participant took 9 hours to answer a question. This is (almost certainly) not representative of the time the participant actually took to answer the question. Therefore, such data had to be removed from the data set before performing statistical analyses and drawing conclusions.

However, we decided that using the total "time spent" on a question was not a good heuristic for cleaning the data. This is because there may be people who genuinely spent a long time on a question, with evidence of continuous modifications. This data point would be legitimate, and removing it could inadvertently skew the data.

Hence, two heuristics were used to determine which data to remove:

1. Did the person work on another question before finishing the current question?
2. Does the longest time interval between code compilations exceed the limit?

Determining if Another Question was Edited We determined if another question was worked on before finishing the current question using the revision timestamps. For each question Q_1, if there was another question Q_2 that had a revision timestamp between the start and end times of Q_1, then Q_1 was removed from the dataset.

The Cut-off of Longest Time Intervals Between Code Compilations Whilst the mean interval length between saved revisions was 8.5 minutes, the median was only 0.4 minutes. Furthermore, 95% of the intervals were below 2 minutes. When looking at the list of the interval times longer than 2 minutes for successfully answered questions (this list can be seen in Appendix D), a relatively continuous increase in intervals can be seen between 2 and 10 minutes, after which a jump to 19 minutes occurs. The cut-off was therefore determined to be 11 minutes, leading to the removal of 4 timing entries.

Per-Question Results for "Correct" Hints

We used the following hypotheses for the times taken with "correct" EEMs:

- **Null hypothesis (H_0):** There is no difference in the time taken to resolve the error between participants who received the EEMs and the OEMs.
- **Alternative hypothesis (H_1):** The participants who receive a "correct" EEM take less time to resolve the error than those who got the OEM.

The timing data for the "correct" hints is visualised in Figure 5.20. Additionally, the mean and median times that were spent on each question can be seen in Table 5.21, and we present the results of the Mann-Whitney U test in Table 5.22.

[2]This is done to increase the ease of reading the results.

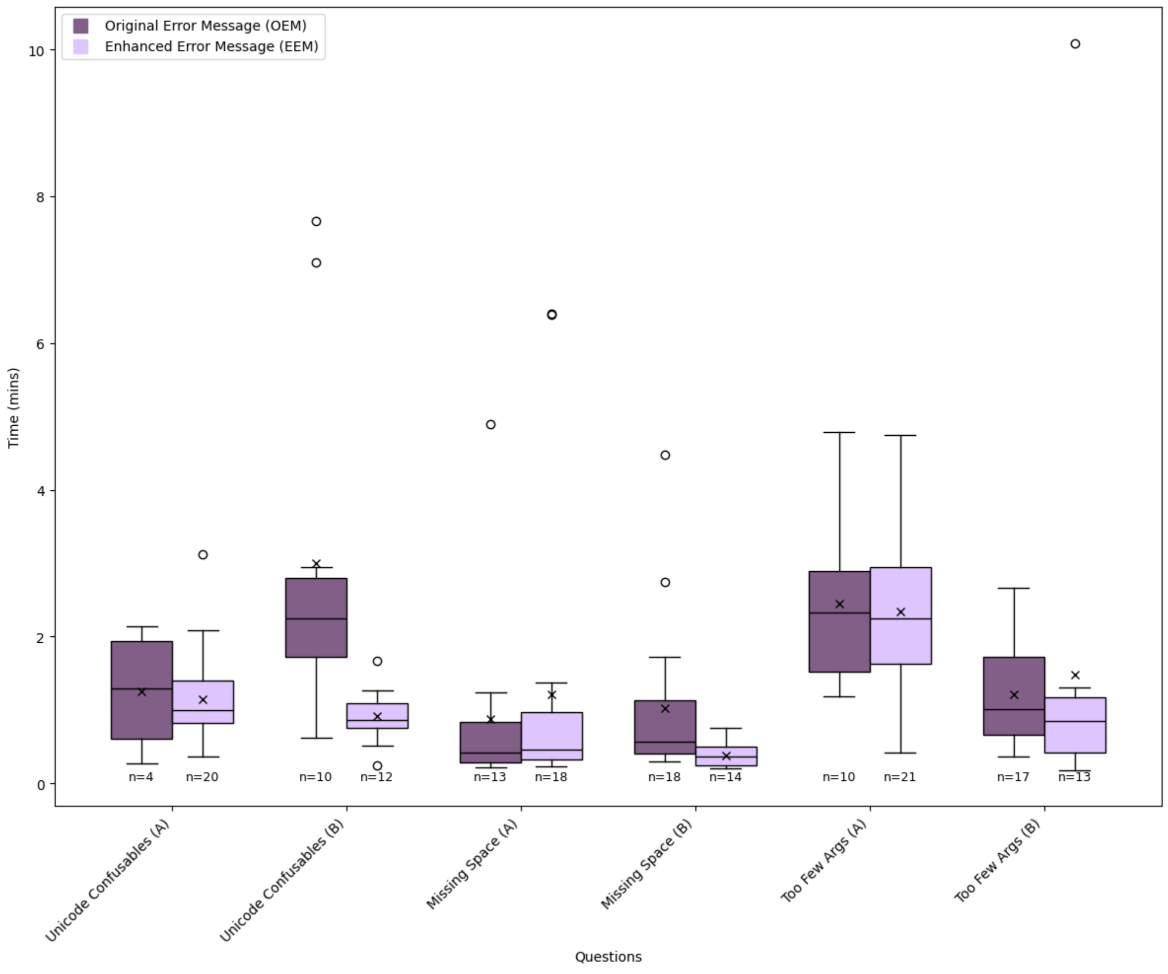


Figure 5.20: Box plot comparing the times spent per question ("Correct" hints)

Question	Mean Time (minutes)			Median Time (minutes)		
	OEM	EEM	Difference (EEM – OEM)	OEM	EEM	Difference (EEM – OEM)
`Unicode Confusables (A)`	1.25	1.14	-0.10	1.29	1.00	-0.29
`Unicode Confusables (B)`	3.00	0.92	-2.08	2.24	0.87	-1.38
`Missing Space (A)`	0.88	1.21	0.33	0.42	0.46	0.04
`Missing Space (B)`	1.02	0.38	-0.64	0.57	0.36	-0.21
`Too Few Args (A)`	2.44	2.33	-0.11	2.33	2.25	-0.08
`Too Few Args (B)`	1.21	1.49	0.28	1.02	0.85	-0.17

Table 5.21: Comparison of the mean and median times spent per question ("Correct" hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Unicode Confusables (A)`	39	0.546	×
`Unicode Confusables (B)`	109	0.001	✓
`Missing Space (A)`	99	0.771	×
`Missing Space (B)`	199.5	0.003	✓
`Too Few Args (A)`	110	0.425	×
`Too Few Args (B)`	138	0.129	×

Table 5.22: Single-tailed MWU results: do "correct" hints decrease the time spent?

From Figure 5.20 and Table 5.21 we can see that the mean and median times spent are mostly smaller for the novices using the EEMs, even though occasional increases are also observed. However, the results of Table 5.22 indicate that there is only evidence in favour of the alternate hypothesis for two of the six programming questions.

Per-Question Results for "Incorrect" Hints

For the "incorrect" hints, we had the following hypotheses:

- **Null hypothesis (H_0):** There is no difference in the time taken to resolve the error between participants who received the EEMs and the OEMs.
- **Alternative hypothesis (H_1):** The participants who receive a "incorrect" EEM take more time to resolve the error than those who got the OEM.

The results are visualised in Figure 5.23, with details of the means and medians written in Table 5.24. Table 5.25 shows the results of our test for statistical significance.

From Figure 5.23 and Table 5.24 we can see that the "incorrect" hint is correlated with a longer mean and median time spent on identifying an error. However, the Mann-Whitney U analysis indicates that this difference is only statistically significant for one `Missing Space` question: for the rest, the null hypothesis is maintained.

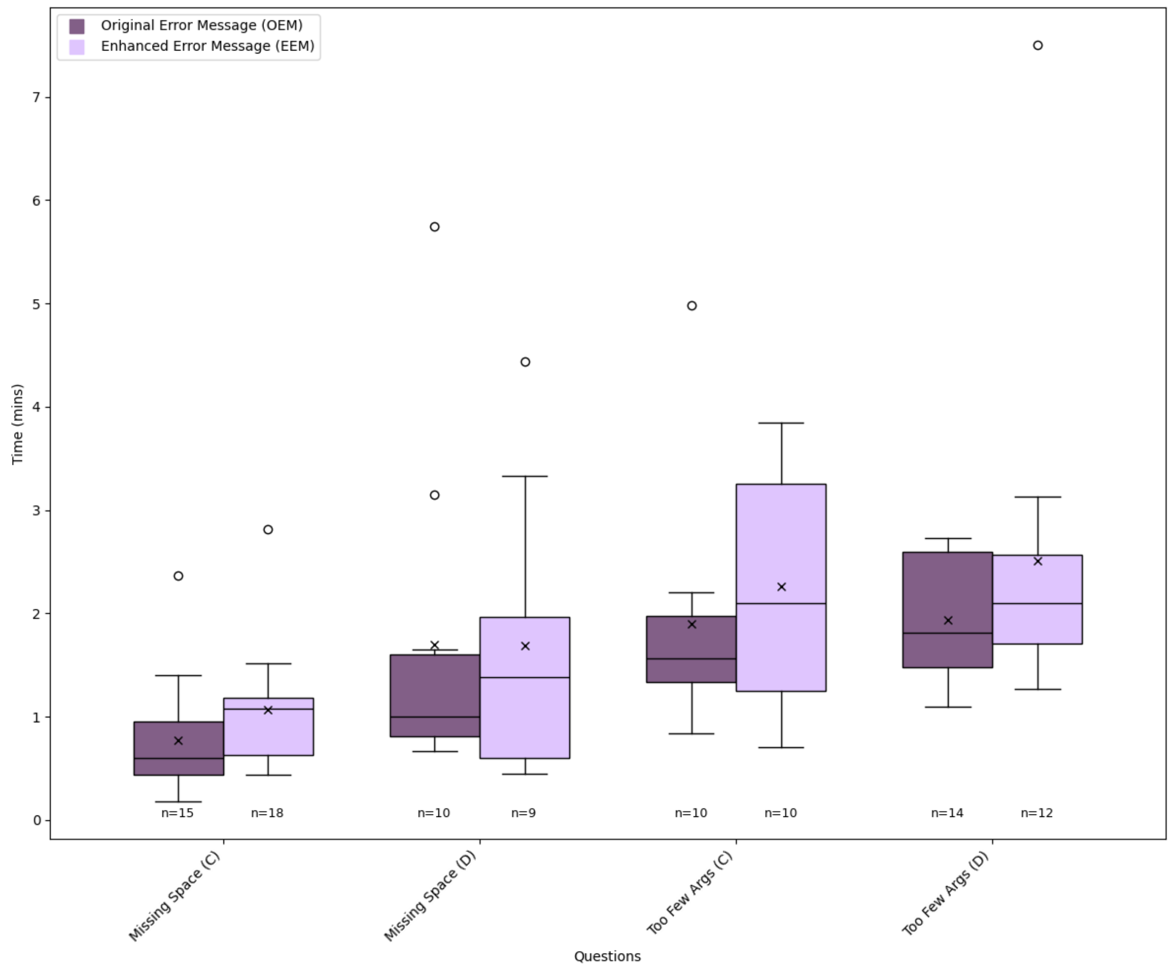


Figure 5.23: Box plot comparing the times spent per question ("Incorrect" hints)

Question	Mean Time (minutes)			Median Time (minutes)		
	OEM	EEM	Difference (EEM – OEM)	OEM	EEM	Difference (EEM – OEM)
`Missing Space (C)`	0.77	1.07	0.30	0.60	1.08	0.48
`Missing Space (D)`	1.70	1.69	-0.01	1.00	1.39	0.38
`Too Few Args (C)`	1.89	2.26	0.37	1.57	2.10	0.53
`Too Few Args (D)`	1.94	2.51	0.57	1.82	2.10	0.28

Table 5.24: Comparison of mean and median times spent per question ("Incorrect" hints)

Question	U Statistic	p Value	Statistical Significance ($p < 0.05$)
`Missing Space (C)`	82.5	0.030	✓
`Missing Space (D)`	46	0.549	×
`Too Few Args (C)`	42.5	0.298	×
`Too Few Args (D)`	72	0.277	×

Table 5.25: Single-tailed MWU results: do "incorrect" hints increase the time spent?

Chapter 6

Discussion

The purpose of this section is to interpret and discuss the results of our user study. Section 6.1 provides such a discussion of our results. Then, we provide a discussion of the limitations of the study in Section 6.2.

6.1 What Does The Data Say?

In order to answer our research questions, we collected both subjective and objective data on the effects of the EEMs. We first discuss our results on the novices' *subjective* perceptions of the error messages in Section 6.1.1. Then, Section 6.1.2 provides a discussion of how the EEMs impacted the novices' objective performance.

6.1.1 Subjective Opinions of the Error Messages

One of the goals of this thesis was to answer **RQ1**: *Which kinds of hints or solutions provided in Agda's error messages are valued by novices?* This section will discuss what conclusion for this research question can be drawn from the results.

The Effect of "Correct" Hints

There is very strong evidence that the EEMs with the "correct" hints are valued higher than OEMs without hints. This was observed both when looking at the results for the EEMs as a whole, and individually. Out of the categories of hints, the strongest evidence was for the Missing Space and the Too Few Args hints.

The weakest evidence for an EEM being perceived as more helpful than the OEM was for `Unicode Confusables (B)`. Although a higher proportion of participants rated it as more helpful than the OEM, this was not statistically significant. This might be partly because the original error message for this programming question was already rated highly —the greatest proportion of participants rated it as "helpful" or "very helpful" out of all the "correct" hints.

The difference in WebLab and agda-mode syntax highlighting might have also influenced the rating of the error message. In Emacs, attempting to type-check the code given in `Unicode Confusables (B)` removes all syntax highlighting aside from colouring the problematic location in red. However, the syntax highlighting in WebLab does not depend on type-checking. Therefore, the different equals signs are highlighted in slightly different colours[1] when viewing the code in WebLab. While this difference is difficult to see in the default WebLab display, it becomes more apparent when using the popular extension Dark Reader[2,3]. This difference can be seen in Table 6.1.

[1]Dark magenta versus black.

[2]`https://darkreader.org/`

[3]Magenta versus white.

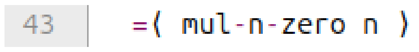
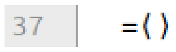
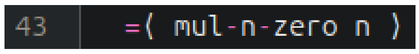
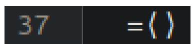
WebLab Display Type	Correct Equals Sign	Incorrect Equals Sign
Default	 `=( mul-n-zero n )`	 `=()`
Dark Reader (dynamic theme generation)	 `=( mul-n-zero n )`	 `=()`

Table 6.1: A comparison of how **=** (U+003D) and ᐀ (U+1400) are presented in WebLab

This could have indicated to programmers that a different equals sign was used on the problematic line, providing similar information to what was given in the hint.

The Effect of "Incorrect" Hints

There is strong evidence that in general, EEMs that presented "incorrect" hints are valued less than the OEMs without the hints. However, it is more difficult to see this difference in valuation for the individual question, as it is only statistically significant for the `Missing Space (D)` question.

Concerningly, the incorrect EEM in `Too Few Args (C)` appears to be higher than the OEM among the participants who could not resolve the error, with 57% of them rating it as "helpful" or "very helpful". This was the programming question where the line seen in Listing 6.1 had to be replaced by Listing 6.2, but the EEM proposes supplying more arguments to `foldr`.

```
intersperse x (y :: ys) = y :: x :: foldr x ys
```

Listing 6.1: Error-containing line from `Too Few Args (C)`

```
intersperse x (y :: ys) = y :: x :: intersperse x ys
```

Listing 6.2: Expected fix to error-containing line in `Too Few Args (C)`

This suggests that the participants were misled by the error message due to unjustified trust in the hint being correct.

Conclusion

To answer **RQ1**, when correct hints are provided by EEMs, they are all valued higher than the OEMs without hints. However, whilst this is supported with the strongest evidence for the Missing Space hints, the evidence is also strongest that the "incorrect" Missing Space EEMs were valued lower than OEMs. Furthermore, incorrect hints are generally valued lower than not having any hint at all.

6.1.2 Novices' Performance

The second research question that the thesis aimed to answer was **RQ2**: *To what extent does providing novices with solutions or hints in Agda's error messages improve their ability to fix errors?* This section will discuss the conclusions made from the results of the success rate and timing data, before giving an overall conclusion about the effects of the EEMs on the novices' performance at solving pre-existing errors.

Success Rate

The success rates of participants were collected and analysed to answer **RQ2.1**: *How does enhancing Agda's error messages with hints influence the probability that novices are able to resolve the error?*

The Effect of "Correct" Hints Overall, there is very strong evidence that EEMs with "correct" hints improve the likelihood that novices are able to resolve the error. Individually, this increase was statistically significant for one question from each of the hint categories:

- `Unicode Confusables (A)`
- `Missing Space (A)`
- `Too Few Args (A)`

When combined with the fact that the success rate was higher for all the participants who got the correct EEMs, this indicates that all of the implemented hints have the potential to increase success rate when correct.

The Effect of "Incorrect" Hints There was no evidence that the "incorrect" EEMs decreased the likelihood of the novice Agda programmers fixing errors in pre-written code.

Conclusion To answer **RQ2.1**, there is strong evidence that, for the implemented hints, enhancing Agda's error messages with hints improves the probability that novice programmers are able to locate and fix the errors without drawbacks.

Timing

The other research sub-question was **RQ2.2**: *How does enhancing Agda's error messages with hints affect the time novices spend fixing the error?* To answer this, programmers were timed on how long they took before successfully completing the programming question. However, only individual per-question conclusions could be drawn from this data due to the differences in sample sizes and time ranges for each question.

The Effect of "Correct" Hints There is strong evidence that the "correct" EEMs the participants took less time to fix the error in two questions:

- `Unicode Confusables (B)`
- `Missing Space (B)`

Curiously, there was no overlap between these and the questions for which a statistically significant increase in success rate was observed with the EEM. This indicates that the two chosen measures of "performance" are not tightly linked.

However, there might be another reason for this discrepancy. All the questions where EEMs led to an increase in success rate were when Group A (those who received Variant A) got the EEM. However, the EEMs led to a decrease in time spent per question only when Group B (those who got Variant B) received the EEM. This might indicate that the discrepancy between success and timing was due to interpersonal differences between the two groups.

For the other questions, there is *no* evidence of EEMs leading to quicker error fixes. In fact, half of the questions led to *higher* mean and/or median times spent resolving the error.

Therefore, whilst EEMs might decrease the time spent fixing an error with certain types of hints, we cannot make the general conclusion that EEMs lead to an improvement in this area.

The Effect of "Incorrect" Hints Only the `Missing Space (C)` programming question bears statistically significant evidence of an incorrect EEM increasing the time spent fixing an error. Therefore, we also cannot make incorrect EEMs lead to more time spent on fixing errors.

Conclusion Unfortunately, it is difficult to draw general conclusions answering **RQ2.2**. However, there is evidence that supports correct EEMs leading to decreases in time spent on questions with `Unicode Confusables` and `Too Few Args` questions, and that incorrect EEMs lead to increases in time spent on `Missing Space` questions.

6.2 Threats to Validity

Although we have obtained valuable insights from the user study, certain limitations must be acknowledged to contextualise the findings. The following section will cover the threats to validity stemming from the recruitment process (see Section 6.2.1), the environment used for the study (see Section 6.2.2), and the creation of the programming questions (see Section 6.2.3). Recognising these limitations will allow for a more nuanced interpretation of the results.

6.2.1 Recruitment

One factor that might decrease the representativeness of the results to the general population of Agda novices were the demographics of the participants. All of the "Agda novices" used in the study were English-speaking[4] students taking the Computer Science & Engineering Bachelors programme at TU Delft[5], a university in the Netherlands. Furthermore, TU Delft has significantly more male students, with only 29.9% of students being female [65]. All of the participants had at least 6 weeks of experience with Haskell, as well as at least 2.5 years of programming in other imperative languages such as Java. The demographics of the participants who took part in the study were further narrowed by most of the recruitment happening during the lecture, with 62% of participants starting their submissions during that time slot. Since lectures were not mandatory, this meant that all of the participants were students who were internally motivated to attend one of the last lectures of the course. Unfortunately, this particular demographic might not be representative of the wider population of Agda novices.

Another problem was that several programming questions included in the user study were inspired by the content of the fourth lecture of the course: "Equational Reasoning in Agda". However, as the lecture was conducted *during* the fourth lecture, this meant that the participants were much less prepared to solve the programming questions on that topic. Four of the ten programming questions were affected:

- `Missing Space (B)`
- `Missing Space (D)`
- `Unicode Confusables (A)`
- `Unicode Confusables (B)`

This amounts to 50% of the `Missing Space` questions, 100% of the `Unicode Confusables` questions, and 0% of the `Too Few Args` questions. This might decrease the validity of the comparisons between the different types of hints.

Additionally, although all the participants took the same course, their levels of experience with Agda might have been different at the time of the study. As the study was held open for 11 days after the final content lecture of the course, the participants who completed the study later likely had more experience with Agda. They would have also likely had more experience

[4]Most of these students were also *not* native English speakers [64]

[5]`https://www.tudelft.nl/en/onderwijs/opleidingen/bachelors/computer-science-and-engineering/bachelor-of-computer-science-and-engineering`

with the equational reasoning questions, as they had the time to do the assignments on the topic. This could result in differences in subjective experiences and objective performance arising from these differences in experience, as opposed to the influence of the error messages.

It should also be noted that since the user study was handed out without monitoring the participants, there is a chance that participants may have collaborated. In that case, the work might be influenced by the error message that their neighbour received. Although this should have been mostly mitigated by the shuffling of the exercises, there was an issue with shuffling on WebLab (this is discussed in Section 6.2.2) that may have increased the impact of this factor on the validity of the results.

6.2.2 Problems with WebLab

Unfortunately, the WebLab exercise shuffling did not work properly during the user study. Whilst all the participants were supposed to receive a shuffled list of questions from the start, in reality most participants received an *un-shuffled* list with `Too Few Args (A)` as the first question. It was only after the submission for that question was opened that the order of the questions was shuffled. This led to `Too Few Args (A)` having a 100% response rate, skewing response rates towards this question significantly.

Another issue that impacts the analysis of the success rate is with the "Abandoned" category. Unfortunately, with the way the timestamps were saved it was impossible to separate people who opened a question with no intention to solve it, versus those who made some sort of attempt before abandoning without saving any changes to the code. This might mean that the real success rate was higher than identified.

WebLab's User Interface

WebLab's User Interface (UI) is another factor that could decrease the representativeness of the results to real-world coding situations. Studies have found that a good user interface can significantly impact the rate at which programmers code [66]. However, WebLab's UI does not provide adequate support for programming in Agda, especially when compared to the recommended editors: Emacs, VsCode, NeoVim, or Vim.

One of the main differences between the recommended WebLab and the recommended text editors is the lack of interactive agda-mode. The main loss is the lack of support for inserting unicode that is not printable ASCII beyond pasting it into the editor. This was problematic for the `Unicode Confusables` category of programming questions, as this meant that the way to "fix" the error was to replace an unusual character with an easy-to-type one. However, this is not representative of the typical Agda experience. As mentioned in Section 3.3, many errors with Unicode confusables must be fixed by doing the opposite. This may reduce the representativeness of the results regarding the UNICODE CONFUSABLES hints.

Furthermore, WebLab uses different syntax highlighting for Agda than the recommended text editors. Normally, the syntax highlighting depends on the type-checking of an Agda file, and if the type-checking fails then the syntax highlighting is not shown[6]. In fact, this is a common complaint with the recommended UI for Agda [25]. However, the syntax highlighting used by WebLab is static, and does not depend on the type-checking. Thus, the highlighting remains even when a type error is present, increasing the readability of the code. This can also lead to differences in typed characters being visible, where they normally wouldn't be, as was discussed in Section 6.1.1. This affects the representativeness of the data obtained from all the programming questions.

While WebLab gives row information for the code, it does not supply the column information. However, the column number is a vital part of the base Agda error messages that

[6]Some editors like Emacs highlight the problem area in red, but others like VS Code lead to no highlighting at all.

communicates the location of the error. Thus, the lack of column information may have decreased the usefulness of all the error messages. For example, in `Unicode Confusables (A)`, there was an error on the following line:

```
+assoc : ∀ (a b c : ℕ) → a + (b + c) ≡ (a + b) + c
```

Both the original and enhanced error messages say that the location of the error is at row 20 and columns 50-51, and that some character `c` is out of scope. However, there are three instances of a character that looks like `c` on the line. When using a text editor that displays the column information, it is easy to identify which `c` is being referred to, but it is significantly harder to do so without it.

However, since the hints in the EEMs sometimes provided additional context information that could be inferred from the column number, like the name of the function that is missing arguments, this could have skewed also the results in favour of the EEMs.

6.2.3 The Programming Questions

In addition to the demographics not being representative of the general Agda novice population, and the UI not being representative of what Agda programmers use, the programming exercises themselves might have been unrepresentative of the errors Agda novices face. The participants of the user study were made to find errors in code that they did not write. They did not know the code as well as they normally would, and did not know what the intentions of the programmer were. Whilst this may simulate debugging someone else's code, this is not a case representative of the standard novice experience. Furthermore, despite attempts at making the errors as organic as possible, they were still created specifically for the user study.

Despite a respectable sample size of participants, there was a limited number of programming questions that tested the effectiveness of the EEMs. There might be other situations where the EEMs are better or worse than the OEMs that the user study did not account for.

Another limitation of the programming questions in the user study was that the setup only allowed us to investigate the detrimental effect of error messages appearing *too frequently*. However, as discussed in Chapter 3, it can also be problematic if an error message does not appear frequently *enough*.

All of this could make it difficult to extrapolate the results of the study to the usefulness of the implemented hints in the real world.

Chapter 7

Related Work

Although dependently-typed proof assistants are valuable tools for both programmers and mathematicians, their usability remains a challenge, particularly for novice users. In contrast to imperative languages, where substantial effort has gone into designing and enhancing error messages to improve programmers' experiences, there is limited research into how proof assistants communicate errors to users. This gap is especially impactful for novice users, who often struggle with cryptic feedback due to lack of experience.

Given the limited research directly addressing the enhancement of error messages in proof assistants, this section draws on several adjacent bodies of work. We begin by reviewing the small but growing literature on the usability of proof assistants, including *proposals* for improving their usability. We then turn to research on the types of errors novice programmers commonly make. Building on this, we examine general recommendations for enhancing error messages, including the use of hints. Finally, we consider existing approaches within proof assistants that aim to improve user feedback.

7.1 Usability of Proof Assistants

Several studies have been conducted on evaluating the usability of proof assistants. One such study, conducted by Kadoda, evaluated 17 different proof assistants (including a dependently-typed proof assistant: LEGO) [67] with Green and Petre's Cognitive Dimensions questionnaire [68]. Based on these results, they proposed a checklist of features that were necessary for a proof assistant with high learnability. The features that were found to have the highest impact on learnability were

- assistance,
- abstraction gradient,
- visibility and juxtaposability,
- perceptual cues,
- meaningful error messages,
- role expressiveness, and
- consistency.

Furthermore, many studies have been done to evaluate the usability of KeY, a (not dependently-typed) formal verification tool for Java [69]. In 2012, Beckert *et al.* used Green and Petre's Cognitive Dimensions questionnaire to evaluate KeY with users that mostly had experience levels varying from "average" to "expert". In this study they found that the most complaints with KeY stemmed from the presentation of information, be that documentation or the proofs themselves, as well as the time performance of KeY. This was followed-up by a study that compared KeY with Isabelle (a higher-order logic (HOL) theorem prover) using focus groups [71]. They identified that convenient interaction with the proof assistant, as well as understandable proof states and processes were desirable traits of proof assistants. Later, Hentschel *et al.* performed a user study that juxtaposed two different user interfaces (UIs) for KeY: the standard KeY UI that had a focus on proof objects, and a Symbolic Execution Debugger (SED) UI that had a view "akin to an interactive debugger". The findings of this study demonstrated the value of a good UI in lowering the barrier to formal verification. This echoes the results of the study conducted by Gray *et al.* in the imperative language Rust, where participants using the "better" UI were 3.3x faster at localising faults than those who used the standard UI [66]. A more recent formative user study with intermediate and expert users of KeY was also carried out in 2020, with the recommendations from the study focusing on proof writing, presentation, and user interaction.

Finally, Juhošová *et al.* have conducted a questionnaire-based user study whose aim was to identify the obstacles that new users of Agda faced [25]. The results showed that novices struggled the most with Agda's ecosystem and IDE design, incomplete supporting material, and "inaccessible support for new users". Particularly relevant for this thesis is that novice Agda programmers appeared to struggle with unclear error messages.

7.2 What Errors do Novice Programmers Make?

Identifying the recurring errors made by novices is important for effectively enhancing error messages, as it allows the enhancements to be made for errors that are actually encountered. However, studies have shown that educators do not have a good understanding of which error messages are actually encountered frequently [73]. Therefore, empirical evidence of error frequencies are vital for identifying which error messages should be enhanced.

We were not able to identify any literature that identified the common errors made by novices of dependently-typed proof assistants. However, such studies have been done for novices of other languages. McCall *et al.* discusses the 10 most common categories of error messages encountered by novices using the BlueJ Java IDE [58], using data from the Blackbox Project [74]. Furthermore, Marie-Hélène Ng Cheong Vee *et al.* investigated the error paths taken by both "novice" and "mature" users of Eiffel, resulting in a list of the common types of errors encountered, as well as the strategies used by novices to deal with the errors they encountered [75]. In a multinational study that used Java and C++[1] found that novice errors stemmed from a "fragile grasp of both basic programming principles and the ability to systematically carry out routine programming tasks". Additionally, in investigating error message frequencies between different languages, Jadud demonstrated the distributions of most common types of errors in BlueJ, COBOL, Helium, LOGO, and SOLO [35]. However, they only provided a list of the most common Java syntax errors.

Unfortunately, it is difficult to apply the results of these studies to enhancing error messages in Agda. The different type systems of these languages mean that the *types* of error messages encountered cannot be accurately mapped to the errors in Agda. Furthermore, the definition of "novices" used in these papers differs from what we mean by Agda novices. Agda is not a common first programming language, with courses usually requiring (or strongly recommending) prior knowledge of a functional programming language like Haskell [76, 77, 78]. However, Java, the most commonly used language in the listed studies, is the second most popular "first programming language" as of 2024 [79]. All of these factors reduce the applicability of the studies to our research.

Although not performed on novices, there has been one study investigating user errors in proof assistants. In investigating HOL proof assistants, Aitken *et al.* provided a model of interaction with proof assistants [80]. They categorised the errors encountered during proofs into

[1]Although the use of language was left up to the participating institution (of the seven total), only these two languages ended up being used.

- syntax errors,
- knowledge errors,
- memory errors, and
- judgements errors.

Although the goal of the study was to provide a taxonomy for ITP errors, this study also looked at the frequency of each category of error, concluding that the programming environment was not a significant source of errors.

7.3 Enhancing Error Messages

A number of other attempts to provide enhanced feedback have been reported in the literature. In 2019, Becker *et al.* provided a comprehensive literature study of the state of error message research, with a focus on error message enhancements [3]. Based on the literature, they presented a list of recommendations, wit historical, anecdotal, and empirical evidence supporting the different types of enhancements:

- Increase Readability,
- Reduce Cognitive Load,
- Provide Context,
- Use a Positive Tone,
- Show Examples,
- Show Solutions or Hints,
- Allow Dynamic Interaction,
- Provide Scaffolding,
- Use Logical Argumentation, and
- Report Errors at the Right Time

In a variety of different user studies of the Decaf Java editor, researchers have consistently showed statistically significant (albeit not overwhelming) support for the enhancements of error messages. In a controlled study, researchers found that enhanced error messages

- reduced the number of errors made by novice programmers, particularly for high-frequency errors,
- reduced the number of struggling students, and
- resulted in more positive experiences with the error messages [8, 2].

This was supported by the results of follow-up studies [9, 10]. In a later study Becker *et al.* found that although the implemented EEMs in Decaf increased the number of errors that got resolve, there was no significant effect on students' scores or the number of compiling submissions [11].

Furthermore, researchers found that enhancing error messages in Athene, a tool for C++, was also correlated with small but statistically significant improvements in novice students' performances in error message quizzes [5]. This was found both in settings with low and high cognitive load.

Nevertheless, the effect of EEMs are not always visible. The study conducted by Denny *et al.* has found no effect (either positive or negative) arising from enhancing Java's syntax error messages [12], and Pettit *et al.* found a similar lack of impact stemming from C++ EEMs [13]. However, they were measuring the number of compiling submissions, for which Becker *et al.* were also unable to find statistically significant differences. Becker *et al.* postulated that this could indicate that the results of these studies are not as contradictory as might initially appear, as "signals" of the effects might be measurable but weak. This could explain why improvements are visible at lower levels (like error frequencies), but not at higher levels (such as compiling submissions).

7.3.1 Empirical Evidence for EEMs with Hints

Unfortunately, research for the error message enhancements mixed hints with other enhancements. This makes it difficult to distinguish the impact that a specific enhancement by itself. Nevertheless, there are some studies that focused their enhancements on hints to the programmer.

In 2010, Hartmann *et al.* implemented hint-based error message enhancements for Java and Arduino, and evaluated the usefulness of the enhancements based on the errors novices made [6]. However, the focus of the study was on *if* the method returned hints that were deemed "useful" by *the authors* of the study. Therefore, they did not evaluate if the novices themselves found the EEMs more useful than the OEMs. Later, Watson *et al.* also used sourced feedback to augment Java's error messages with hints. However, the study also incorporated other error message enhancements, such as elaborating the feedback and presenting dynamic levels of support based on the programmer's behaviour [7]. Nevertheless, the findings suggested that novice students perceived the EEMs as significantly more helpful than the OEMs. A study with enhancing error messages with hints in Python also found that the hints were useful in high-certainty situations, when the hint is very unlikely to present incorrect information, as they "contributed to [novice] students" progress while still encouraging the students to solve problems by themselves" [14].

7.4 Improving Error Messages in Functional Languages

Although much in improving and enhancing error messages has been done in imperative languages, the same cannot be said for functional languages, especially dependently-typed proof assistants. However, this does not mean that there is *no* research on this topic. This section will briefly cover the recent efforts that have gone into enhancing error messages in purely functional languages like Haskell, or even proof assistants like Agda.

One such project that aims at enhancing error messages for novice programmers is Helium, an alternate Haskell compiler. In his PhD. thesis, Heeren details the principles that were used in the generation of error messages in Helium, which used a constraint-based type-inference algorithm [81]. Later, Chen *et al.* discussed a method for counter-factual type-inference algorithm that could generate more informative error messages with suggestions. They evaluated both the usefulness and efficiency of the approach with a prototype in Haskell [82]. Furthermore, Eremondi *et al.* proposed a method that used replay graphs and counter-factual solving to improve the error messages in dependently-typed languages [24].

However, none of this research evaluated the usefulness of the enhanced error messages with actual users. Such user studies are vital to measure the actual impacts that these "enhancements" had on the usability of the language(s) [70]. The lack thereof therefore indicates the gap in the research that our thesis aimed to fill.

Chapter 8

Conclusion

The goal of this thesis was to investigate the effects of enhancing Agda's error messages with hints on Agda novices: their performances on programming exercises, and their opinions on the helpfulness of the hints. To do this, we implemented hint enhancements to three error messages:

- MISSING SPACE: forgetting whitespace between names and/or operators,
- UNICODE CONFUSABLES: accidentally using a Unicode confusable character, and
- TOO FEW ARGS: supplying a function with too few arguments.

To investigate the usefulness of these EEMs, we then performed a user study that validated

- novices' perceptions of the helpfulness of hint,
- the time taken for novice Agda programmers to find a resolve an error in pre-written code, and
- their success rate in finding and resolving the errors.

Out of the hints for Agda's error messages implemented for the user study, they were all found to increase satisfaction with the helpfulness of the error message, and rate of success in fixing the error. Furthermore, while the hints that displayed incorrect suggestion were occasionally rated as more misleading than the original error messages without the hints, the incorrect information did not have any impact on the probability that they could fix the error. The evidence for the correct or incorrect enhanced error messages affecting the time spent resolving a question was inconclusive.

In conclusion, Agda's error messages that were enhanced with hints were both valued better by novice Agda programmers, and increased novices abilities to fix errors. We can therefore recommend both further enhancing error messages in dependently-typed proof assistants like Agda with hints, and conducting more research into this topic.

8.1 Advice for User Studies

We recommend doing a longitudinal study on the impacts of the enhanced error messages. For example, if investigating their effects on novices, this could be done over the period of an entire course on some dependently-typed proof assistant. Although this type of investigation has been done to investigate the impact of EEMs on imperative languages [13, 9], we were not able to identify such longitudinal studies in dependently-typed proof assistants. In addition to error message usability, this would also give the opportunity to investigate the effect of EEMs on the language's learnability:

- Do people who *learnt* the language with the EEMs perform better on exams?

- Do they make *fewer* errors?

The errors encountered by the participants of such a study would be more representative of errors faced by all novices, as they are not artificially made. Furthermore, doing such a user study on EEMs with hints would also allow the trade-off between showing a hint too frequently and too infrequently to be investigated.

Since it is highly unlikely for all the hints that could provide useful information to *always* point in the correct direction, we also believe that it would be valuable to investigate how often the correct / incorrect suggestions are actually displayed to properly judge the extent to which the effect of incorrect error messages is detrimental.

8.2 Further Work

In addition to our advice for future user studies in enhancing error messages, we also provide recommendations for future work.

8.2.1 Does Enhancing Error Messages Help *Experienced* Programmers?

Our study, as well as most studies into the usability of EEMs [3], focused on the impacts of EEMs on *novices'* experiences, but they are not the only programmers who could benefit from enhanced error messages. EEMs could potentially reduce cognitive load, thereby increasing the usability of the language even for experienced users. Therefore, we believe that investigating the effects of EEMs on a wider population of proof assistant programmers may be beneficial.

8.2.2 Which Error Messages Should Be Enhanced?

In our research, we selected the error messages based on anecdotal evidence for common novice errors. Despite there being empirical evidence of error message frequencies for imperative languages like Python and Java, as well as proof assistants that are *not* dependently typed, there hasn't been any research into this for *dependently-typed* proof assistants (to the extent of our knowledge). An empirical study into this could therefore be valuable, as it would help guide further work by giving concrete evidence of which errors are common and should be improved.

Furthermore, it may be the case that common errors are *not* the errors that need to be focused on in EEM research. Research by McCall *et al.* has identified that error frequency may be a poor indicator for the errors that novice programmers struggle with [58]. Therefore, an investigation of the *severity* of errors in dependently-typed proof assistants may be warranted, to ensure that the greatest impact on novice usability is attained.

8.2.3 Other Methods of Enhancing Error Messages for Dependently-Typed Proof Assistants

The heuristics that we used in this thesis to determine when the hint should be presented were selected due to being easy to implement, while still allowing to test the usefulness of *providing* such a hint. However, they were not always the best heuristics. This was particularly the case for the implemented Too Few Args hint, where it would be presented at the wrong time very frequently. Future work can focus on implementing *and* investigating the usability of more algorithms for enhancing error messages, like what was suggested by Eremondi, Swierstra, and Hage [24].

Furthermore, while we investigated the impacts of enhancing error messages with hints, this is not the only type of enhancement that can be made. There are other recommended enhancements to error messages that could be valuable to investigate, such as those compiled by Becker *et al.* [3]:

- Increase Readability,
- Reduce Cognitive Load,
- Provide Context,
- Use a Positive Tone,
- Show Examples,
- Allow Dynamic Interaction,
- Provide Scaffolding,
- Use Logical Argumentation, and
- Report Errors at the Right Time.

Investigating (and comparing) the effects of these enhancements on the usability of dependently-typed proof assistants can be a valuable avenue of research. This may be particularly apt for allowing dynamic interaction, as programming in many proof assistants (such as Agda) is already interactive.

Bibliography

[1] T. Barik, J. Smith, K. Lubick, *et al.*, "Do Developers Read Compiler Error Messages?" In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 575–585. DOI: `10.1109/ICSE.2017.59`.

[2] B. A. Becker, "An Effective Approach to Enhancing Compiler Error Messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16, New York, NY, USA: Association for Computing Machinery, Feb. 17, 2016, pp. 126–131, ISBN: 978-1-4503-3685-7. DOI: `10.1145/2839509.2844584`.

[3] B. A. Becker, P. Denny, R. Pettit, *et al.*, "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*, ser. ITiCSE-WGR '19, New York, NY, USA: Association for Computing Machinery, Dec. 18, 2019, pp. 177–210, ISBN: 978-1-4503-7567-2. DOI: `10.1145/3344429.3372508`.

[4] G. Marceau, K. Fisler, and S. Krishnamurthi, "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*, ser. Onward! 2011, New York, NY, USA: Association for Computing Machinery, Oct. 22, 2011, pp. 3–18, ISBN: 978-1-4503-0941-7. DOI: `10.1145/2048237.2048241`.

[5] J. Prather, R. Pettit, K. H. McMurry, *et al.*, "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, New York, NY, USA: Association for Computing Machinery, Aug. 14, 2017, pp. 74–82, ISBN: 978-1-4503-4968-0. DOI: `10.1145/3105726.3106169`.

[6] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, New York, NY, USA: Association for Computing Machinery, Apr. 10, 2010, pp. 1019–1028, ISBN: 978-1-60558-929-9. DOI: `10.1145/1753326.1753478`.

[7] C. Watson, F. W. B. Li, and J. L. Godwin, "BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair," in *Advances in Web-Based Learning - ICWL 2012*, E. Popescu, Q. Li, R. Klamma, H. Leung, and M. Specht, Eds., Berlin, Heidelberg: Springer, 2012, pp. 228–239, ISBN: 978-3-642-33642-3. DOI: `10.1007/978-3-642-33642-3_25`.

[8] B. A. Becker, "An Exploration Of The Effects Of Enhanced Compiler Error Messages For Computer Programming Novices," Technological University Dublin, Nov. 1, 2015.

[9] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Computer Science Education*, vol. 26, no. 2–3, pp. 148–175, Jul. 2, 2016, ISSN: 0899-3408. DOI: 10.1080/08993408.2016.1225464.

[10] B. A. Becker and C. Mooney, "Categorizing Compiler Error Messages with Principal Component Analysis," May 29, 2016. [Online]. Available: http://hdl.handle.net/10197/7889 (visited on 05/30/2025).

[11] B. A. Becker, K. Goslin, and G. Glanville, "The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18, New York, NY, USA: Association for Computing Machinery, Feb. 21, 2018, pp. 640–645, ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159461.

[12] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14, New York, NY, USA: Association for Computing Machinery, Jun. 21, 2014, pp. 273–278, ISBN: 978-1-4503-2833-3. DOI: 10.1145/2591708.2591748.

[13] R. S. Pettit, J. Homer, and R. Gee, "Do Enhanced Compiler Error Messages Help Students? Results Inconclusive.," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17, New York, NY, USA: Association for Computing Machinery, Mar. 8, 2017, pp. 465–470, ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017768.

[14] P. M. Phothilimthana and S. Sridhara, "High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?" In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17, New York, NY, USA: Association for Computing Machinery, Jun. 28, 2017, pp. 182–187, ISBN: 978-1-4503-4704-4. DOI: 10.1145/3059009.3059058.

[15] Emillie Thiselton and Christoph Treude, "Enhancing Python Compiler Error Messages via Stack," presented at the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019. DOI: 10.1109/ESEM.2019.8870155.

[16] A. Bove and P. Dybjer, "Dependent Types at Work," in *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, Eds., Berlin, Heidelberg: Springer, 2009, pp. 57–99, ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_2.

[17] H. Xi, "Dead Code Elimination through Dependent Types," in *Practical Aspects of Declarative Languages*, G. Gupta, Ed., Berlin, Heidelberg: Springer, 1998, pp. 228–242, ISBN: 978-3-540-49201-6. DOI: 10.1007/3-540-49201-1_16.

[18] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Berlin, Heidelberg: Springer, 2009, pp. 73–78, ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_6.

[19] X. Leroy, S. Blazy, Z. Dargaye, *et al.* "CompCert - Context and motivations." (Jul. 4, 2023), [Online]. Available: https://compcert.org/motivations.html (visited on 06/12/2025).

[20] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, p. 20 150 399, Sep. 4, 2017. DOI: `10.1098/rsta.2015.0399`.

[21] U. Norell, "Dependently Typed Programming in Agda," in *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, P. Koopman, R. Plasmeijer, and D. Swierstra, Eds., Berlin, Heidelberg: Springer, 2009, pp. 230–266, ISBN: 978-3-642-04652-0. DOI: `10.1007/978-3-642-04652-0_5`.

[22] J. Harrison, J. Urban, and F. Wiedijk, "History of Interactive Theorem Proving," in *Handbook of the History of Logic*, ser. Computational Logic, J. H. Siekmann, Ed., vol. 9, North-Holland, Jan. 1, 2014, pp. 135–214. DOI: `10.1016/B978-0-444-51624-4.50004-6`.

[23] E. W. Dijkstra, "The Humble Programmer," *Commun. ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1, 1972, ISSN: 0001-0782. DOI: `10.1145/355604.361591`.

[24] J. Eremondi, W. Swierstra, and J. Hage, "A framework for improving error messages in dependently-typed languages," *Open Computer Science*, vol. 9, no. 1, pp. 1–32, Jan. 1, 2019, ISSN: 2299-1093. DOI: `10.1515/comp-2019-0001`.

[25] S. Juhošová, A. Zaidman, and J. Cockx, "Pinpointing the Learning Obstacles of an Interactive Theorem Prover," 2025.

[26] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "QED at Large: A Survey of Engineering of Formally Verified Software," *Foundations and Trends® in Programming Languages*, vol. 5, no. 2–3, pp. 102–281, 2019, ISSN: 2325-1107, 2325-1131. DOI: `10.1561/2500000045`. arXiv: `2003.06458 [cs]`.

[27] "Unicode - APL Wiki." (), [Online]. Available: `https://aplwiki.com/wiki/Unicode` (visited on 02/07/2025).

[28] "Unicode Input · The Julia Language." (), [Online]. Available: `https://docs.julialang.org/en/v1/manual/unicode-input/` (visited on 02/07/2025).

[29] "Unicode | Raku Documentation." (), [Online]. Available: `https://docs.raku.org/language/unicode` (visited on 02/07/2025).

[30] E. Allen, D. Chase, J. Hallett, *et al.*, "The Fortress language specification," *Sun Microsystems*, vol. 139, no. 140, p. 116, 2005. [Online]. Available: `http://www.eecis.udel.edu/~cavazos/cisc879-spring2008/papers/fortress.pdf` (visited on 02/07/2025).

[31] David Tolnay, workingjubilee, est31, Jieyou Xu, and Noratrieb. "Warn on Fullwidth Exclamation Mark (U+FF01) in comment · Issue #134810 · rust-lang/rust," GitHub. (Dec. 2024), [Online]. Available: `https://github.com/rust-lang/rust/issues/134810` (visited on 05/07/2025).

[32] 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*, ser. ESEC/FSE 2018, New York, NY, USA: Association for Computing Machinery, Oct. 26, 2018, pp. 633–643, ISBN: 978-1-4503-5573-5. DOI: `10.1145/3236024.3236040`.

[33] V. J. Traver, "On Compiler Error Messages: What They Say and What They Mean," *Advances in Human-Computer Interaction*, vol. 2010, no. 1, p. 602 570, 2010, ISSN: 1687-5907. DOI: `10.1155/2010/602570`.

[34] "Command-line options — Agda 2.7.0.1 documentation." (), [Online]. Available: `https://agda.readthedocs.io/en/v2.7.0.1/tools/command-line-options.html` (visited on 05/06/2025).

[35] M. C. Jadud, "An exploration of novice compilation behaviour in BlueJ," Ph.D. dissertation, University of Kent, 2006, 268 pp. doi: `10.22024/UniKent/01.02.86458`.

[36] "Data.Product.Base," Documention for the Agda standard library. (), [Online]. Available: `https://agda.github.io/agda-stdlib/v2.2/Data.Product.Base.html` (visited on 04/17/2025).

[37] "Lexical Structure — Agda 2.7.0.1 documentation." (), [Online]. Available: `https://agda.readthedocs.io/en/v2.7.0.1/language/lexical-structure.html#keywords-and-special-symbols` (visited on 04/17/2025).

[38] "Algebra.Properties.Group." (), [Online]. Available: `https://agda.github.io/agda-stdlib/v2.2/Algebra.Properties.Group.html` (visited on 04/17/2025).

[39] "Names," The Agda Wiki. (Dec. 14, 2018), [Online]. Available: `https://wiki.portal.chalmers.se/agda/ReferenceManual/Names` (visited on 04/18/2025).

[40] "Word Break II," LeetCode. (2025), [Online]. Available: `https://leetcode.com/problems/word-break-ii/description` (visited on 05/06/2025).

[41] Heman Gandhi. "How not to Learn Agda." (), [Online]. Available: `https://hemangandhi.github.io/blog-posts/agda.html` (visited on 01/17/2025).

[42] "Warning against Unicode confusables - Internals & Design," Julia Programming Language. (Jan. 9, 2024), [Online]. Available: `https://discourse.julialang.org/t/warning-against-unicode-confusables/108734` (visited on 02/07/2025).

[43] Michael Nahas, Jesper Cockx, Andreas Abel, and Nils Anders Danielsson. "Unicode identifier visually similar to ASCII one (source of confusion/obfuscation) · Issue #5629 · agda/agda," GitHub. (Nov. 21), [Online]. Available: `https://github.com/agda/agda/issues/5629` (visited on 05/07/2025).

[44] gkepka, S. Layland, A. Korsakov, *et al.* "Collections Types," Scala Documentation. (), [Online]. Available: `https://docs.scala-lang.org/scala3/book/collections-classes.html` (visited on 06/13/2025).

[45] Rocq. "Library Corelib.Init.Datatypes," Rocq Core Library. (), [Online]. Available: `https://rocq-prover.org/doc` (visited on 06/13/2025).

[46] "Emacs Mode — Agda 2.7.0.1 documentation." (), [Online]. Available: `https://agda.readthedocs.io/en/v2.7.0.1/tools/emacs-mode.html#ok-but-how-can-i-find-out-what-to-type-to-get-the-character` (visited on 05/07/2025).

[47] zeithaste. "Unicode code point of current character - Visual Studio Marketplace." (May 8, 2024), [Online]. Available: `https://marketplace.visualstudio.com/items?itemName=zeithaste.cursorCharCode` (visited on 06/01/2025).

[48] Josip Medved. "Unicode Code Point - Visual Studio Marketplace." (May 25, 2025), [Online]. Available: `https://marketplace.visualstudio.com/items?itemName=medo64.code-point` (visited on 06/01/2025).

[49] Frederik Hanghøj Iversen, Andreas Abel, and G. Allais. "Suggest replacements for various unicode characters · Issue #2898 · agda/agda," GitHub. (Jan. 2018), [Online]. Available: `https://github.com/agda/agda/issues/2898` (visited on 05/07/2025).

[50] Unicode, Inc. "Unicode Utilities: Confusables." (), [Online]. Available: `https://util.unicode.org/UnicodeJsps/confusables.jsp?a=a&r=None` (visited on 06/01/2025).

[51] Ben Hamilton. "Data.Text.ICU.Spoof," Hackage. (2015), [Online]. Available: `https://hackage.haskell.org/package/text-icu-0.8.0.5/docs/Data-Text-ICU-Spoof.html` (visited on 05/07/2025).

[52] X. Lee. "Emacs: Insert Unicode Character." (Sep. 2, 2024), [Online]. Available: `http://xahlee.info/emacs/emacs/emacs_n_unicode.html` (visited on 06/01/2025).

[53] L. Chonavel. "Move unicode keybindings into JSON file by lawcho · Pull Request #7094 · agda/agda," GitHub. (Feb. 7, 2024), [Online]. Available: `https://github.com/agda/agda/pull/7094` (visited on 03/03/2025).

[54] L. Chonavel, N. A. Danielsson, and J. Marozas. "Generate a '.XCompose' file · Issue #7083 · agda/agda," GitHub. (May 2, 2024), [Online]. Available: `https://github.com/agda/agda/issues/7083` (visited on 06/01/2025).

[55] agda. "Later · Milestone," GitHub. (), [Online]. Available: `https://github.com/agda/agda/milestone/38` (visited on 06/01/2025).

[56] Philip Wadler, Wen Kokke, and Jeremy G. Siek. "Programming Language Foundations in Agda – Getting Started." (), [Online]. Available: `https://plfa.github.io/GettingStarted/` (visited on 02/06/2025).

[57] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '03, New York, NY, USA: Association for Computing Machinery, Jan. 11, 2003, pp. 153–156, ISBN: 978-1-58113-648-7. DOI: `10.1145/611892.611956`.

[58] D. McCall and M. Kölling, "A New Look at Novice Programmer Errors," *ACM Trans. Comput. Educ.*, vol. 19, no. 4, 38:1–38:30, Jul. 11, 2019. DOI: `10.1145/3335814`.

[59] "Brightspace," TU Delft. (), [Online]. Available: `https://www.tudelft.nl/en/student/my-study-me/study-tools/brightspace` (visited on 05/09/2025).

[60] M. Khakimova, J. Cockx, S. Juhošová, and J. Reinders, *Enhancing Proof Assistant Error Messages with Hints: A User Study (dataset)*, 4TU.ResearchData, 2025. DOI: `10.4121/79e7c4eb-81dc-492a-9ac4-69f33166de8e`.

[61] N. Nachar, "The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution," *Tutorials in Quantitative Methods for Psychology*, vol. 4, no. 1, 2008. DOI: `10.20982/tqmp.04.1.p013`.

[62] R. A. Fisher, "Statistical Methods for Research Workers," in *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N. L. Johnson, Eds., New York, NY: Springer, 1992, pp. 66–70, ISBN: 978-1-4612-4380-9. DOI: `10.1007/978-1-4612-4380-9_6`.

[63] H. Aguinis, M. Vassar, and C. Wayant, "On reporting and interpreting statistical significance and p values in medical research," *BMJ Evidence-Based Medicine*, vol. 26, no. 2, pp. 39–42, Apr. 2021, ISSN: 2515-446X. DOI: `10.1136/bmjebm-2019-111264`. PMID: `31732498`.

[64] M. Van Der Veldt. "TU Delft students speak more than three languages," Delta. (Feb. 26, 2019), [Online]. Available: `https://delta.tudelft.nl/en/article/tu-delft-students-speak-more-three-languages` (visited on 06/15/2025).

[65] "Facts and Figures," TU Delft. (), [Online]. Available: `https://www.tudelft.nl/en/about-tu-delft/strategy/diversity-inclusion/facts-and-figures-1` (visited on 04/15/2025).

[66] G. Gray, W. Crichton, and S. Krishnamurthi. "An Interactive Debugger for Rust Trait Errors." arXiv: `2504.18704 [cs]`. (Apr. 25, 2025), pre-published.

[67] G. F. Kadoda, "Formal software development tools: An investigation into usability," Ph.D. dissertation, Loughborough University, Jan. 1, 1997. [Online]. Available: `https://repository.lboro.ac.uk/articles/thesis/Formal_software_development_tools_an_investigation_into_usability/9407444/1` (visited on 05/05/2025).

[68] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, Jun. 1, 1996, ISSN: 1045-926X. DOI: `10.1006/jvlc.1996.0009`.

[69] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book* (Lecture Notes in Computer Science). Cham: Springer International Publishing, 2016, vol. 10001, isbn: 978-3-319-49811-9 978-3-319-49812-6. doi: `10.1007/978-3-319-49812-6`.

[70] B. Beckert and S. Grebing, "Evaluating the usability of interactive verification systems," presented at the CEUR Workshop Proceedings, vol. 873, 2012, pp. 3–17.

[71] B. Beckert, S. Grebing, and F. Böhl, "A Usability Evaluation of Interactive Theorem Provers Using Focus Groups," in *Software Engineering and Formal Methods*, C. Canal and A. Idani, Eds., Cham: Springer International Publishing, 2015, pp. 3–19, isbn: 978-3-319-15201-1. doi: `10.1007/978-3-319-15201-1_1`.

[72] M. Hentschel, R. Hähnle, and R. Bubel, "An empirical evaluation of two user interfaces of an interactive program verifier," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, New York, NY, USA: Association for Computing Machinery, Aug. 25, 2016, pp. 403–413, isbn: 978-1-4503-3845-5. doi: `10.1145/2970276.2970303`.

[73] N. C. C. Brown and A. Altadmri, "Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs," *ACM Trans. Comput. Educ.*, vol. 17, no. 2, 7:1–7:21, May 3, 2017. doi: `10.1145/2994154`.

[74] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, New York, NY, USA: Association for Computing Machinery, Mar. 5, 2014, pp. 223–228, isbn: 978-1-4503-2605-6. doi: `10.1145/2538862.2538924`.

[75] Marie-Hélène Ng Cheong Vee, Bertrand Meyer, and Keith L. Mannock. "Empirical study of novice errors and error paths." (2005), pre-published.

[76] "CSE 3100: Functional Programming," Course browser searcher. (), [Online]. Available: `https://www.studiegids.tudelft.nl/a101_displayCourse.do?course_id=67585` (visited on 05/30/2025).

[77] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78, isbn: 978-3-642-03358-2 978-3-642-03359-9. doi: `10.1007/978-3-642-03359-9_6`.

[78] "Advanced Programming Paradigms." (), [Online]. Available: `https://www.msengineering.ch/theory-modules/2024-2025-tsm-advprpa` (visited on 05/30/2025).

[79] JetBrains Academy. "Computer Science Learning Curve Survey 2024 Report," JetBrains. (2024), [Online]. Available: `https://lp.jetbrains.com/cs-learning-curve-report-2024/` (visited on 05/29/2025).

[80] S. Aitken and T. Melham, "An analysis of errors in interactive proof attempts," *Interacting with Computers*, vol. 12, no. 6, pp. 565–586, Jul. 2000, issn: 09535438. doi: `10.1016/S0953-5438(99)00023-5`.

[81] B. J. Heeren, "Top quality type error Messages," Sep. 20, 2005. [Online]. Available: `https://dspace.library.uu.nl/handle/1874/7297`.

[82] S. Chen and M. Erwig, "Counter-factual typing for debugging type errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, New York, NY, USA: Association for Computing Machinery, Jan. 8, 2014, pp. 583–594, isbn: 978-1-4503-2544-8. doi: `10.1145/2535838.2535863`.

Acronyms

HCI	Human-Computer Interaction
EEM	Enhanced Error Message
OEM	Original Error Message
MWU	Mann-Whitney U

Appendix A

Full List of Identified Hints in Agda

The following appendix displays all of the error message hints (without the base error message) that were found after manually investigating all occurrences of `parseError` and analysing the following files:

- `Agda.Interaction.Options.Errors`
- `Agda.Syntax.Concrete.Definitions.Errors`
- `Agda.TypeChecking.Errors`
- `Agda.TypeChecking.Serialise.Instances.Errors`
- `Agda.Interaction.Options.Warnings`
- `Agda.TypeChecking.Monad.Base.Warning`
- `Agda.TypeChecking.Warnings`
- `Agda.TypeChecking.Pretty.Warning`

The appendix contains the following groups of hints:

- Directions (see Appendix A.1),
- Suggestions (see Appendix A.2), and
- Questions (see Appendix A.3).

A.1 Directions

A.1.1 `Agda.TypeChecking.Errors`

- `NoBindingForBuiltin`
  - `use {-# BUILTIN` [**SUGGESTION**][1] `name #-} to bind builtin natural numbers to the type 'name'`
  - `use {-# BUILTIN` [**SUGGESTION**]`name #-} to bind it to 'name'`
- `ModuleDefinedInOtherFile`
  - `However, according to the include path this module should be defined in` [**SUGGESTION**]
- `ClashingDefinition`
  - `In data definitions separate from data declaration, the ':' and type must be omitted.`
- `NeedOptionPatternMatching`

[1]Note: [**SUGGESTION**] here is used as a placeholder for variables that are determined at compile time

  - (use option --pattern-matching to enable it)

- NeedOptionProp
  - (use options --prop and --no-prop to enable/disable Prop)
- NeedOptionTwoLevel
  - (use option --two-level to enable SSet)
- NeedOptionUniversePolymorphism
  - (use option --universe-polymorphism to allow level arguments to sorts)
- AttributeKindNotEnabled
  - (use [SUGGESTION] to enable them): [SUGGESTION]
- NamedWhereModuleInRefinedContext
  - See https://github.com/agda/agda/issues/2897.
- PatLamWithoutClauses
  - Use a single 'absurd-clause' for absurd lambdas

A.1.2 Agda.TypeChecking.Pretty.Warning

- CoinductiveEtaRecord
  - If you must, use pragma {-# ETA [SUGGESTION] #-}
- UnusedVariablesInDisplayForm
  - Replace it by an underscore to pacify this warning
  - Replace them by underscores to pacify this warning

A.1.3 Agda.Syntax.Concrete.Definitions.Errors

- pretty
  - If the opaque definitions are to be mutually recursive, move the 'mutual' block inside the 'opaque' block.
  - Since [SUGGESTION] can not participate in mutual recursion, their definition must be given before this point.
- PragmaNoTerminationCheck
  - To skip the termination check, label your definitions either as {-# TERMINATING #-} or {-# NON_TERMINATING #-}

A.2 Suggestions

A.2.1 Agda.TypeChecking.Errors

- CannotEliminateWithPattern b p a
  - (suggestion: write [SUGGESTION] for a dot pattern)
- CantGeneralizeOverSorts
  - Suggestion: add a 'variable Any : Set _' and replace unsolved metas by Any

- `ModuleNameUnexpected`
  - `The module` [SUGGESTION] `should probably be named` [SUGGESTION]
- `ModuleNameHashCollision`
  - `(you may want to consider renaming one of these modules)`
- `AmbiguousModule`
  - `(hint: Use C-c C-w (in Emacs) if you want to know why)`
- `DotPatternInPatternSynonym`
  - `Maybe use '_' instead.`
- `CannotResolveAmbiguousPatternSynonym`
  - `(hint: overloaded pattern synonyms must be equal up to variable and constructor names)`
- `ImpossibleConstructor`
  - `Possible solution: remove the clause, or use an absurd pattern ().`
- `NotAHaskellType`
  - `Possible fix: add a pragma` [SUGGESTION] `for a suitable Haskell` [SUGGESTION] `.`
  - `Possible fix: replace the value-level pragma at` [SUGGESTION] `by` [SUGGESTION] `for a suitable Haskell` [SUGGESTION] `.`
- `SolvedButOpenHoles`
  - `(consider adding {-# OPTIONS --allow-unsolved-metas #-} to this module)`

A.2.2 **`Agda.TypeChecking.Pretty.Warning`**

- `IllegalRewriteRule`
  - `You can turn off the projection-like optimization for` [SUGGESTION] `with the pragma {-# NOT_PROJECTION_LIKE` [SUGGESTION] `#-} or globally with the flag --no-projection-like`
- `InversionDepthReached`
  - `Most likely this means you have an unsatisfiable constraint, but it could also mean that you need to increase the maximum depth using the flag --inversion-max-depth=N`
- `RewriteAmbiguousRules`
  - `Possible fix: add a rewrite rule with left-hand side` [SUGGESTION] `to resolve the ambiguity.`
- `RewriteMissingRule`
  - `Possible fix: add a rule to rewrite`

A.2.3 **`Agda.Syntax.Concrete.Definitions.Errors`**

- `NotAllowedInMutual`
  - `Suggestion: get rid of the mutual block by manually ordering declarations`

A.2.4 parseError

- Agda.Syntax.Parser.LexActions > lexToken
  - (you may want to replace tabs with spaces)

A.3 Questions

A.3.1 Agda.TypeChecking.Errors

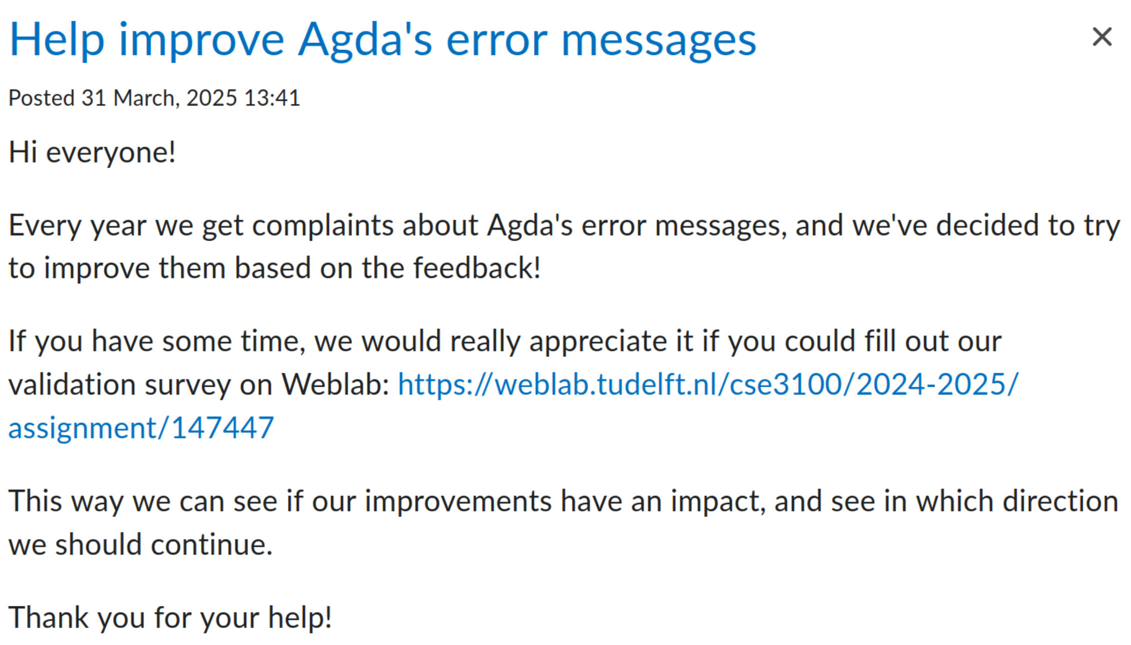
- CannotEliminateWithPattern b p a
  - (did you supply too many arguments?)
- ClashingDefinition
  - Perhaps you meant to write [SUGGESTION] at [SUGGESTION] ?
- IllTypedPatternAfterWithAbstraction
  - (perhaps you can replace it by '_'?)

A.3.2 Agda.TypeChecking.Pretty.Warning

- IllegalRewriteRule
  - Perhaps you can use a postulate instead of a constructor as the head symbol?
- NotInScopeW
  - (did you forget space around the ':'?)
  - (did you forget space around the '->'?)
  - (did you mean [SUGGESTION] ?)

Appendix B

User Study: Brightspace Announcement



Help improve Agda's error messages

Posted 31 March, 2025 13:41

Hi everyone!

Every year we get complaints about Agda's error messages, and we've decided to try to improve them based on the feedback!

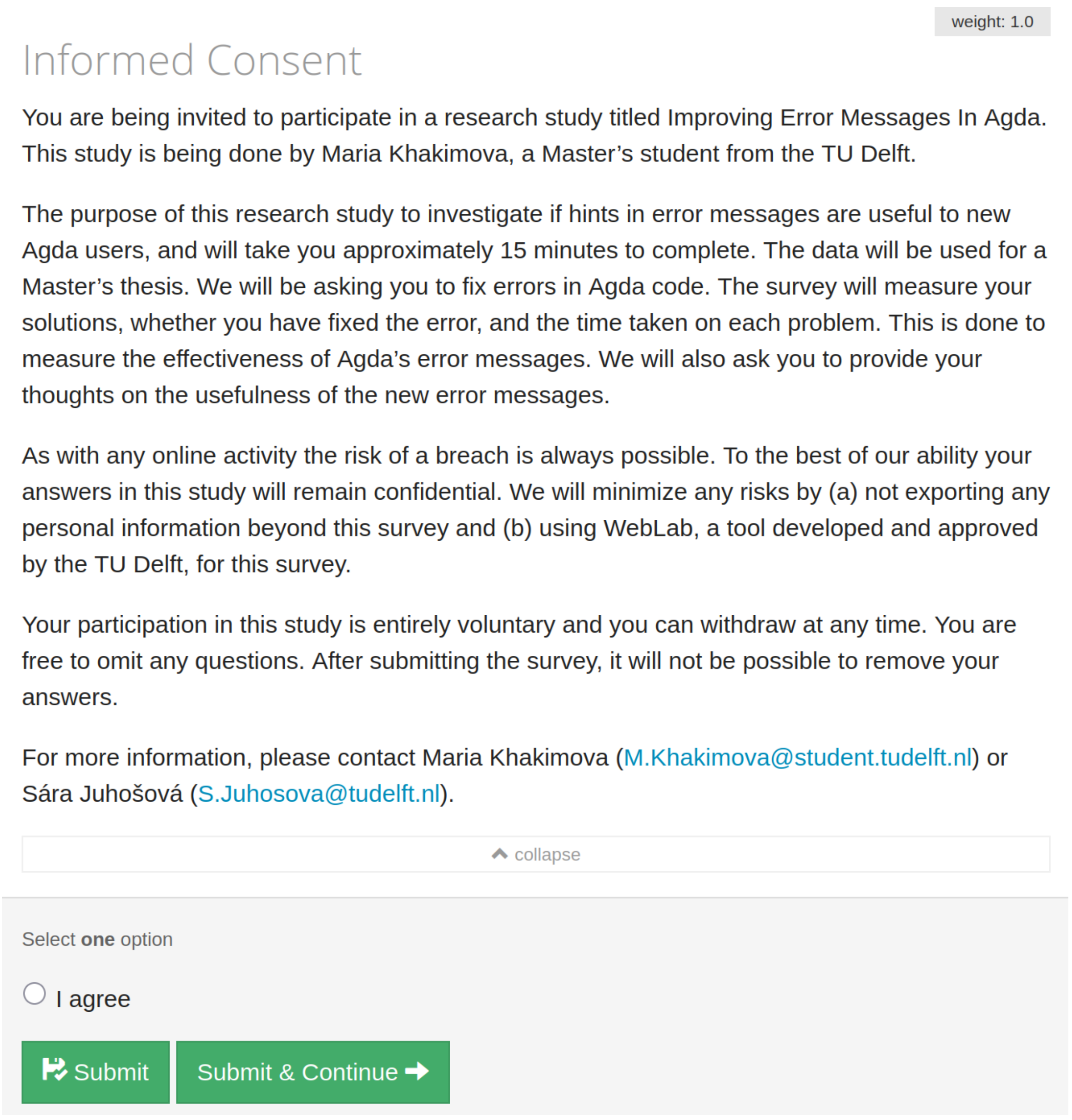
If you have some time, we would really appreciate it if you could fill out our validation survey on Weblab: https://weblab.tudelft.nl/cse3100/2024-2025/assignment/147447

This way we can see if our improvements have an impact, and see in which direction we should continue.

Thank you for your help!

Appendix C

User Study: Informed Consent



weight: 1.0

Informed Consent

You are being invited to participate in a research study titled Improving Error Messages In Agda. This study is being done by Maria Khakimova, a Master's student from the TU Delft.

The purpose of this research study to investigate if hints in error messages are useful to new Agda users, and will take you approximately 15 minutes to complete. The data will be used for a Master's thesis. We will be asking you to fix errors in Agda code. The survey will measure your solutions, whether you have fixed the error, and the time taken on each problem. This is done to measure the effectiveness of Agda's error messages. We will also ask you to provide your thoughts on the usefulness of the new error messages.

As with any online activity the risk of a breach is always possible. To the best of our ability your answers in this study will remain confidential. We will minimize any risks by (a) not exporting any personal information beyond this survey and (b) using WebLab, a tool developed and approved by the TU Delft, for this survey.

Your participation in this study is entirely voluntary and you can withdraw at any time. You are free to omit any questions. After submitting the survey, it will not be possible to remove your answers.

For more information, please contact Maria Khakimova (M.Khakimova@student.tudelft.nl) or Sára Juhošová (S.Juhosova@tudelft.nl).

Select **one** option

○ I agree

Appendix D

Results: "Longest Intervals" Longer than 2 Minutes

This appendix shows a complete list of the longest intervals within a submission in descending order. This excludes the intervals from submissions that were removed due to other heuristics detailed in Section 5.4.2.

- 5587.43333333333 mins
- 568.4 mins
- 554.583333333333 mins
- 60.9166666666667 mins
- 34.7166666666667 mins
- 19.35 mins
- 10.0833333333333 mins
- 7.66666666666667 mins
- 7.1 mins
- 6.4 mins
- 6.38333333333333 mins
- 5.08333333333333 mins
- 4.18333333333333 mins
- 3.65 mins
- 3.63333333333333 mins
- 3.43333333333333 mins
- 3.41666666666667 mins
- 2.98333333333333 mins
- 2.93333333333333 mins
- 2.9 mins
- 2.7 mins
- 2.65 mins
- 2.6 mins
- 2.51666666666667 mins
- 2.48333333333333 mins

- 2.45 mins
- 2.43333333333333 mins
- 2.33333333333333 mins
- 2.31666666666667 mins
- 2.3 mins
- 2.26666666666667 mins
- 2.25 mins
- 2.23333333333333 mins
- 2.21666666666667 mins
- 2.2 mins
- 2.15 mins
- 2.13333333333333 mins
- 2.1 mins
- 2.08333333333333 mins
- 2.08333333333333 mins
- 2.03333333333333 mins
- 2.01666666666667 mins