Enhancing Interactive Theorem Prover Error Messages with Hints

Maria Khakimova
Delft University of Technology, The Netherlands

Sára Juhošová
Delft University of Technology, The Netherlands

Jaro Reinders
Delft University of Technology, The Netherlands

Jesper Cockx
Delft University of Technology, The Netherlands

Abstract

Interactive theorem provers (ITPs) are promising tools for ensuring program correctness, but users often complain about their poor usability and steep learning curve. A common complaint, especially among new users, are confusing error messages that expose details of the ITP's underlying theory or implementation details. In this work, we investigate how adding hints to three types of scope and type checking error messages in the Agda ITP affects the new users' debugging experience. We evaluate the effectiveness and perceived helpfulness of those error messages by conducting a between-subjects user study where we provide a series of Agda code snippets, each containing a single error that the participants have to fix based on the error message. We measure the success rate, time taken to fix the error, and perceived helpfulness for each code snippet with the original as well as the enhanced error message and determine the statistical significance of adding the hint. Our results show that *correct* hints can improve the success rate and time taken to fix the error, and that error messages with hints are rated significantly more helpful than those without. Additionally, we find that while error messages with *incorrect* hints are often rated as more misleading, they do not significantly impact the success rate or time taken to fix the error. These results show that adding hints to error messages is a viable step on the path towards making ITPs more widely accessible.

2012 ACM Subject Classification Human-centered computing → Empirical studies in interaction design

Keywords and phrases Agda, error messages, hints, new users

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.5

Supplementary Material *Dataset*: `https://doi.org/10.4121/10425039-1e9c-46d2-80c9-a2805c1b7e62` [26]

Declaration about GenAI/LLM use The only use of generative AI / LLMs in this work was via an activated GitHub Copilot plugin in the Pycharm editor, used when analysing the data.

1 Introduction

Designed as programming languages with type systems strong enough to allow static program verification, dependently-typed interactive theorem provers (ITPs) such as Agda [32], Coq/Rocq [38], and Lean [14] can produce software that is completely verified with respect to its specification. This capability should, theoretically, make ITPs *the* choice tool for programmers implementing critical software components, but we are yet to see their spread into mainstream development processes. The most prominent obstacles that new users face are poor usability and steep learning curves, with previous research showing that the languages'

ecosystems do not provide adequate support for their users [23]. New users in particular claim to struggle with confusing error messages which "require theoretical knowledge and experience to be helpful" [22, p. 166].

Encountering error messages is a major part of programming, with Java programmers spending 13%–25% of their time reading compiler error messages [4]. The feedback that error messages provide aids programmers in understanding the roots of their errors and brings them closer to a working program, which is especially important for new users who have limited experience with the language. Confusing error messages can cause students to lose confidence [7], demotivate potential users from learning the language, or even lead programmers away from the correct solution [28]. Such error messages are known to cause significant frustration among new users, and are frequently considered to be a "barrier to progress" [6]. We therefore anticipate that improving ITP error message helpfulness could reduce perceived barriers for new users and make ITPs more accessible to a wider range of programmers.

In a literature review on the effectiveness of error messages and their enhancements, Becker et al. [7] provided ten ways to enhance error messages:

1. increase readability,
2. reduce cognitive load,
3. provide context,
4. use a positive tone,
5. show examples,
6. show solutions or hints,
7. allow dynamic interaction,
8. provide scaffolding,
9. use logical argumentation, and
10. report errors at the right time.

There are two main research gaps in existing literature on enhancing ITP error messages (discussed in detail in Section 6) that are relevant to our work:

1. there is limited research that isolates the effect of *one* enhancement on the usability of error messages in general, and
2. none of the research focusing on error messages in ITPs conducts a *user study* to evaluate the effects of the improved error messages.

To close these two gaps, we enhanced an ITP's error messages with one recommendation from the guidelines above, and evaluated the obtained enhanced error messages (EEMs) against the original error messages (OEMs) with a user study.

Out of the potential enhancements, we chose to provide hints or solutions to the programmer. Although there is a lot of empirical evidence that supports the positive impact of hints in error messages [41, 34, 19, 5], Marceau et al. warn against providing hints because they can lead users "down the wrong path" [28]. Furthermore, despite the fact that papers providing empirical evidence for showing solutions or hints were most common in Becker et al.'s comprehensive literature review [7], all those studies combined the addition of hints with other enhancements. Therefore, it is difficult to isolate the effect that enhancing error messages with hints has on the usability of the error messages.

In this paper, we answer the following main research question:

RQ: How do hints in ITP error messages affect their usability for new users?

Firstly, we want to understand the practical influence of hints on the new users' ability to fix errors. However, we also want to investigate whether hints can change how users *perceive* such error messages – after all, we wish to eliminate confusing error messages from the *perceived* obstacles they face when learning to use an ITP. As such, we use the following sub-questions to answer our main research question:

RQ1 How do hints in ITP error messages affect the ability of new users to fix errors?
- **RQ1.1** How do hints affect the success rate of new users fixing errors?
- **RQ1.2** How do hints affect the time new users spend fixing an error?

RQ2 How do new users perceive the helpfulness of hints in ITP error messages?

To answer these research questions, we chose to enhance three error messages in Agda's scope and type checker. Agda is often used in computer science education and provides a good opportunity for studying the effect of its error messages on users who are new to ITPs but not necessarily new to programming. We make the following contributions in this work:

- a design for adding hints to three error messages in Agda's scope and type checker;
- a user study design for understanding the effect of those hints, with all resources provided in our data repository under an MIT license [26]; and
- a detailed analysis of how hints influence the usability of ITP error messages for new users.

Our results show that *correct* hints can improve the success rate and time taken to fix the error, and that error messages with hints are rated significantly more helpful than those without. Additionally, we find that while error messages with *incorrect* hints are often rated as more misleading, they do not significantly impact the success rate or time taken to fix the error. As a result, we conclude that enhancing ITP error messages with hints is a viable way to improve their usability and recommend adding a hint where deemed potentially useful.

2 Hint Design

To investigate whether hints in ITP error messages are useful for new users, we first designed hints for some of Agda's existing error messages. To pick which error messages to enhance, we turned to those identified as learning obstacles [22] and chose ones which we believe are idiosyncratic to Agda or ITPs in general. We focused on designing three enhanced error messages (EEMs) on top of the original error messages (OEMs) from Agda `v2.8.0`:

- WHITESPACE: forgetting whitespace between variables or operators;
- CONFUSABLE: using the wrong, visually confusable Unicode character; and
- TOOFEWARGS: not passing enough arguments to a function.

In the following section, we explain the general considerations that guided our design and then dive into the specific implementation of each enhanced error message.

One of the important principles in how *all* error messages are presented is *consistency* [40, p. 10]. Consistency serves to reduce cognitive load on the programmers, allowing them to more efficiently process the information provided by the compiler [7]. 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.

To identify hints within Agda's existing error messages, we needed to define what a "hint" is. For the purpose of this work, we used the following definition:

- a hint has to be an *addition* to the base error message, and
- a hint needs to provide a (potential) solution to the programmer.

Even with these guidelines, hints can come in many shapes and sizes: the order in which information is presented can vary; the hint can be phrased as a question, suggestion, or direction; and the framing phrases can vary (e.g., "`possible fix: ...`" or "`you may want to ...`").

Though it was impossible to identify *all* the messages with hints in Agda's codebase (consisting of over 185K lines of source code), looking at just the `Error` and `Warning` modules of the project already showed that there is a lot to be desired in terms of hint consistency within the current pool of error messages. We therefore looked at hints in *similar* error messages to the ones we were planning to implement. The only existing hints we found similar to the first two errors were phrased as questions, presented in parentheses, and framed using the "`did you ...?`" phrase (e.g., "`(did you forget space around the ':'?)`"). Similarly, the error message for passing *too many* (as opposed to *too few*) arguments to a function had the following hint appended to the end: "`(did you supply too many arguments?)`". As a result, we design all our enhanced error messages to use this parenthesised phrasing.

Ideally, each hint would be presented only in situations when applying the fix suggested by the hint would "fix" the code. This could be done by, for example, having Agda apply the proposed fix and rerunning the type checker before presenting the EEM. However, given the limitations of the Agda compiler, we were not able to do this. Currently, Agda's compiler stops type checking upon encountering *an* error, meaning that it is only ever aware of one error at a time. As a result, if the rerun were to encounter another error, we would have no reliable way of determining whether it is caused by the applied "fix" or by some unrelated part of the code. Therefore, within this project, we chose to display the hints according to some heuristic (described in the following subsections). This means that some error messages might display irrelevant hints, leading programmers "down the wrong path" (as Marceau et al. warned [28]).

Whitespace

The first type of error we enhanced with a hint is the `[NotInScope]` error thrown when the programmer forgets to add spaces around mixfix operators [37] – an error identified to be a common obstacle for new Agda users [22]. This error arises because Agda allows programmers to define operators by adding underscores to function names. For example, when a function `_&&_ : Bool → Bool → Bool` is defined, it can be called in three ways:

- by applying the arguments in the standard way: `_&&_ True False`;
- by applying the arguments infix: `True && False`; or
- by only applying *some* of the arguments infix: `(_&& False) True`.

However, the infix arguments *must* be separated from the operator with a whitespace, since Agda will consider anything not separated by a whitespace as one name. For example, if we would write `True&& False`, Agda would think we are trying to apply the function named `True&&` to `False` (which will throw a `[NotInScope]` error, since it does not exist).

To help guide users, especially the ones that are not used to Agda's syntax, we added a hint telling them they might have forgotten to add a whitespace *when the not-in-scope name can be broken down completely into names that* are *in scope*. Note, though, that not all the options provided would actually solve the problem – some only satisfy the scope checker but not necessarily the type checker (e.g., `pancake` might be split into `pan` and `cake` if both are in scope, but applying `pan` to `cake` might not result in a value of the type that was expected in that position). The following demonstrates an example of the WHITESPACE enhanced error message, with the assumption that variables `sn` and `d` are also in scope:

```
swap : A × B → B × A
swap (fst , snd) = (snd, fst)
```

(lines with "+" contain the enhancement)

```
    /home/me/examples/swap.agda:2.21-25: error: [NotInScope]
    Not in scope:
    snd, at /home/me/examples/swap.agda:2.21-25
+       (did you forget whitespace in
+           'snd ,' or
+           'sn d ,'?)
    when scope checking snd,
```

Confusable

Agda supports the use of Unicode characters in function and variable names, which allows Agda proofs to look very similar to paper proofs, but "raises the barrier of entry" for new users and creates confusion during program comprehension [22, p. 165]. Since the set of Unicode characters contains many "confusables" (characters that are visually similar or identical to each other), it can often be difficult to understand an error telling you that "`=`" is not the same as "=" (a problem that gets worse with certain fonts). This is a relatively common complaint from the students learning Agda at our university, but also users on the internet [20]. Similar complaints have been bought up in other languages that offer Unicode support, such as in Julia [1].

Since Unicode characters are ubiquitously used in Agda and Agda's libraries, it is no surprise that efforts have already been made to provide support to users for this scenario. In the `agda-mode` plugin for the Emacs editor, users can use a command to get information about the character the cursor is positioned over [36]. However, the programmer needs to both know that this command exists, and think to check 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 and programmers who use a different development environment (e.g., VS Code[1] or Vim) will not have easy access to this information.

The best way to provide this information such that the code can be fixed easily is through a hint in the error message that is caused by precisely such a mismatch. The addition of such a hint has also been suggested several times by Agda users [16, 29], but is yet to be implemented by the Agda developers. For the design of this hint, we took inspiration from the way Rust [27] presents these types of errors, and 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 `agda-mode`.

The hint is displayed whenever a name is not in scope and there is at least one name in scope in which all the characters (in order) are either exactly the same or confusable. To determine whether characters are confusable, we used the `text-icu` package[2] that uses the `confusablesWholeScript.txt`[3] file for spoof-checking [12]. The `agda-mode` Unicode input key bindings are drawn from a list compiled by Chonavel [13]. The following demonstrates an example of the Confusable enhanced error message:

[1] VS Code does have extensions that can identify a Unicode character [42, 21], but no references are made to them in the Agda documentation.
[2] `https://hackage.haskell.org/package/text-icu-0.8.0.5`
[3] `http://unicode.org/Public/security/latest/confusablesWholeScript.txt`

```
id : Set → Set
id bαp = baρ
```

(lines with "+" contain the enhancement)

```
    /home/me/examples/alpha.agda:2.10-13: error: [NotInScope]
    Not in scope:
    baρ at /home/me/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)  MATH 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
```

There are a few limitations to this solution:

- **The definition of which characters are confusable is not Agda-specific.** There are many characters that are important to Agda programmers that are not covered by `confusablesWholeScript.txt` (e.g., "<" and "⟨").
- **Confusability is only checked against names in scope.** There is also a chance that the programmer used a confusable character instead of a *reserved name*, in which case our implementation does not add the hint.
- **Unicode characters with a non-standard length cause misalignment of the `diff` in the error message.** Some Unicode characters have a different size than the standard, even in monospace font, and can misalign the `diff` strings and arrows.
- **The circumstances in which the EEM is shown are limited.** For example, if a name has both a typo and a confusable error, the current implementation will not display the EEM.

We suggest that these limitations be addressed before adding the EEM to Agda. The current implementation is, however, sufficient to help us answer our research question.

TooFewArgs

The final error message we enhanced was an `[UnequalTerms]` type checking error which arises when too few arguments are passed to a function. For simplicity, we only provide the name of the function in the hint. A more sophisticated hint could also include the location of the function definition and the types of the missing arguments.

Unfortunately, due to polymorphism and type inference it is not always clear when too few arguments are supplied to a function call. In our design, the EEM is shown when the error appears at a function call site and the inferred type of the function application is a function type. This does mean that, for example, when the programmer does not supply the function with *any* arguments, the hint will not appear, since the error is not at the "call site" of a function. The following demonstrates an example of the TooFewArgs enhanced error message:

```
addThree : Nat → Nat → Nat → Nat
addThree a b c = a + b + c
num : Nat
num = addThree 1 2
```

(lines with "+" contain the enhancement)

```
    /home/me/examples/args.agda:4.7-19: 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 Study Setup

To answer our research questions, we conducted a between-subjects user study on a class of final-year bachelor students learning to use Agda for the first time. We designed code snippets with errors they had to find and fix, collecting the **timestamp and compilation status of each compilation attempt**, as well as their **opinion on the helpfulness of the error message** using a five-point Likert scale. We describe the participants, study design, and data processing in the following section. The study was conducted with the approval of the human research ethics committee of our university.

3.1 Context & Participants

The participants in this study were final-year computer science bachelor students taking an elective Functional Programming course at our university. The course held lectures over seven weeks, consisting of ten lectures on functional programming in Haskell and four Agda lectures with the goal of teaching students to

- interactively develop Agda programs,
- use the Curry-Howard correspondence [35, p. 108] to express logical properties as types,
- use indexed data types and dependent pattern matching to enforce invariants of their programs, and
- formally prove properties of purely functional programs by using the identity type and equational reasoning.

The students were introduced to these topics during live lectures and were given programming exercises to practice on. This was the first time the students came into contact with an ITP and, as such, could be labelled "new users". They were encouraged to use the `agda-mode` extension in the Visual Studio (VS) Code editor. However, their work was submitted, compiled, and verified through submission to an online learning management system developed at our university (described in Section 3.2.3).

3.2 Study Design

To determine whether hints improve the developer experience of new users when fixing errors, we designed a between-subjects experiment with ten debugging assignments for each participant to attempt. Each assignment featured

- a code snippet that contained *exactly one* compilation error, simulating being in the middle of writing or debugging the code, and
- a follow-up question, asking participants to rate the helpfulness of the error message on a five-point Likert scale.

The participants' task was to run the compiler on the code snippet, read the error message, and enter a cycle of attempting to fix the error and recompile the code until it succeeded. We then used the status and timestamp of each compilation attempt to answer **RQ1** and the Likert scale ratings to answer **RQ2**.

3.2.1 Code Snippet

We designed the study such that each participant would receive one debugging assignment for each "correct" EEM (i.e., offering the hint that will actually fix the error). Because Whitespace and TooFewArgs can also appear in incorrect situations (i.e., they are hinting at something that will not fix the error), we added an assignment for both a *correct* and

incorrect "error message intention" for these two hint types. The CONFUSABLE hint appears only in very specific situations, and we could not find a sufficiently realistic scenario to provide for an *incorrect* situation. We deemed it unnecessary to include such an assignment in the study.

Furthermore, we wanted participants to encounter each error message in both original and enhanced form, but did not want them to see the same code twice. That would give them a chance to use prior experience with the code to fix the error, thus skewing the results. Therefore, we designed two assignments for each possible hint type and error message intention and used a between-subjects design to distribute two variants of the survey. For example, assignment TOOFEWARGS#A used the following code snippet and EEM:

```
data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)

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

updateElement : {A B : Set}{n : Nat}
    → Vec A n → Fin n → B → (A → B → A) → Vec A n
updateElement (x :: xs) zero    b f = f x :: xs
updateElement (x :: xs) (suc i) b f = x :: updateElement xs i b f
```

(lines with "+" contain the enhancement)

```
  /home/student/solution.agda:12.39-42: error: [UnequalTerms]
  (B → A) !=< A
  when checking that the inferred type of an application
    B → A
  matches the expected type
    A
+ (did you supply too few arguments to f ?)
```

Similarly, TOOFEWARGS#B was designed to have the same style of error, and was provided to a participant with the opposite type of error message to TOOFEWARGS#A:

```
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 _++_

zipWith : {A B C : Set}{n : Nat}
  → (A → B → C) → Vec A n → Vec B n → Vec C n
zipWith f []       []       = []
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs
```

(lines with "+" contain the enhancement)

```
  /home/student/solution.agda:14.42-54: error: [UnequalTerms]
  (Vec _B_24 _n_26 → Vec _C_25 _n_26) !=< Vec C n
  when checking that the inferred type of an application
    Vec _B_24 _n_26 → Vec _C_25 _n_26
  matches the expected type
    Vec C n
+ (did you supply too few arguments to zipWith ?)
```

An overview of the assignments in each variant is available in Table 1. In order to keep consistent with the experience level of our participants, the code snippets were inspired by existing exercises from the course. This meant that we did not use the Agda standard library,

Table 1 An overview of the debugging assignments used in the study, including the "expected" fix for each assignment and the line on which that fix had to be applied.

intention	error message	tag	in variant A	in variant B	error line	fix
correct	WHITESPACE	`#A`	EEM	OEM	24/24	`(x, y)` → `(x , y)`
		`#B`	OEM	EEM	23/25	`~b` → `~ b`
	CONFUSABLE	`#A`	EEM	OEM	20/22	cyrillic "c" → latin "c"
		`#B`	OEM	EEM	37/45	syllabics hyphen "=" → equals sign "="
	TOOFEWARGS	`#A`	EEM	OEM	13/14	`f x` → `f x b`
		`#B`	OEM	EEM	15/15	`zipWith f xs` → `zipWith f xs ys`
incorrect	WHITESPACE	`#C`	EEM	OEM	23/23	`xs,i` → `xs i`
		`#D`	OEM	EEM	44/54	`+-id` → `+-identity`
	TOOFEWARGS	`#C`	EEM	OEM	18/18	`foldr x ys` → `intersperse x ys`
		`#D`	OEM	EEM	24/24	`f (f a) a` → `f b a`

instead providing the needed functionality in student-visible "library" code. The syntax of this "library" code also uses fewer Unicode characters than the Agda standard library, since the online environment we used does not support Emacs-style Unicode insertion methods. Additionally, some exercises were adapted from the Agda standard library[4], the Iowa Agda library[5], and the *Programming Language Foundations in Agda*[6] book. All assignments are publicly available in our data repository [26], including the code, the accompanying original and enhanced error messages, and the expected "corrected" code. For quick reference, Table 1 contains an overview of the fix each assignment required.

3.2.2 Likert Scale Ratings

Each debugging assignment was concluded by a multiple-choice question with a five-point Likert scale, asking participants to rate the helpfulness of the provided error message on a scale of "very helpful" to "very misleading". We opted to include the neutral "neither helpful nor misleading". A sixth "(N/A): Did not read the error message" was added to differentiate from the neutral option, and to help us eliminate responses which did not focus on the object of study (the error message).

[4] `https://agda.github.io/agda-stdlib/`
[5] `https://github.com/cedille/ial`
[6] `https://plfa.github.io/`

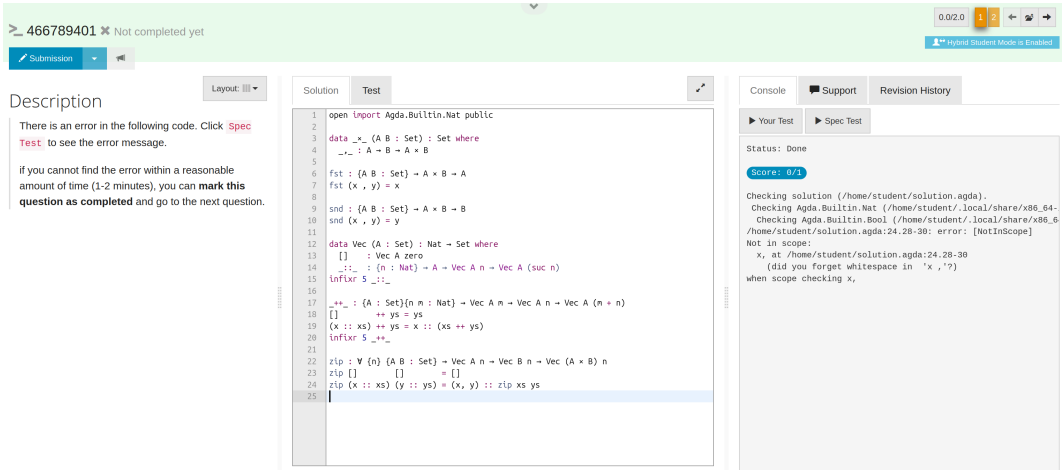

Figure 1 The online environment interface displaying the WHITESPACE#A assignment.

3.2.3 Programming Environment

We used an online system which allows courses to build exercises in a single environment. Importantly, it allows for the creation of programming assignments, which students can edit, compile, and test directly in the environment. By using this system, we were able to

- use two different versions of Agda (one with the original and one with the enhanced error messages), without having to ask the participants to install and configure them,
- randomise which variant each participant got,
- randomise the order in which each participant got the assignments, and
- collect the timestamp and status of each compilation attempt in one system.

Additionally, the students were familiar with the system from this and previous courses, minimising the influence that the programming environment had on our results. Figure 1 shows the interface of a programming assignment in the system from a students' perspective.

3.3 Pilot & Distribution

Before distributing the survey, we conducted a pilot on four people (two for each variant) with varying levels of Agda experience. The purpose of this pilot was to identify any technical issues with the environment setup, check for spelling mistakes, ensure the difficulty was at the right level, and get an estimate of how long the full survey would take to complete. Three of the pilot participants had more experience with Agda than the target participants, and one was a new Agda user.

After looking at the pilot participants' statistics and conducting an exit interview with them, we judged the difficulty to be appropriate but the duration to be surprisingly long. On average, the more experienced participants took 16 minutes, while the new Agda user took 35 minutes. Most of the delay was caused by spending too much time on an error they were not able to fix. Based on these findings, we added a line to the description of each assignment urging participants to continue on to the next assignment if they could not complete it "within a reasonable amount of time (1-2 minutes)".

We distributed the survey during the last lecture of the course as well as via an announcement posted on the main communication page of the course. The students were given ten minutes of lecture time to complete the survey, with the option to continue into the 15-minute mid-lecture break. Alternatively, they could complete it on their own time in the eleven-day window during which it was available.

3.4 Data Cleaning

Because some participants only opened the survey but did not actually participate, we had to differentiate their response from the valid ones. We removed a participant (and their submissions) from our dataset when the data showed they attempted only a single assignment and their "submission" to that assignment contained only a single timestamp (i.e., the first compilation) and did not have a rating for the perceived helpfulness. We reasoned that though this data *could* be relevant to the success rate, we had no way of distinguishing that from participants who simply got distracted or lost interest when they saw the format of the study. This removed a total of 17 participants (and their 17 single submissions).

Additionally, since the user study was open for eleven days, and no strict order to the assignments was enforced, participants were free to stop debugging at any point and return to the assignment later. While this likely increased participation rates per assignment, it occasionally resulted in data saying that a participant took, e.g., 9 hours to debug a code snippet. This is (almost certainly) not true. We removed timing data for a submission from our dataset in three cases:

1. **If the participant did not successfully fix the error:** This removed 136 submissions (responses to an individual assignment) from our timing analysis.
2. **If the participant worked on another assignment before finishing the current submission:** For each submission S_1, if there was another submission S_2 from the same participant that had a compilation timestamp between the start and end times of S_1, S_1 was marked invalid. This removed an additional 9 submissions from our timing analysis.
3. **If the longest time interval between code compilations was unreasonable:** We estimated that the longest reasonable time a new user would spend between compilations is two minutes (this is also the time we instructed the participants to consider "reasonable" before moving on to the next assignment). As a sanity check, we ran the analysis both on data where all submissions with an interval larger than two minutes were cleaned, and on data where all submissions with an interval larger than five minutes were cleaned. There was no statistically significant difference between the two. As such, we left the two-minute cut-off and removed an additional 25 submissions from our timing analysis.

3.5 Data Analysis

To test for significance in the differences between the enhanced and original error messages, we used single-tailed Mann-Whitney U tests. This is a statistical test to "compare two independent groups that do not require large normally distributed samples" [30, p. 13], which made it an appropriate choice for this study. 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.

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 allow us to test if EEMs are better or worse than OEMs, depending on whether the error message offered a *correct* or *incorrect* hint. All null and alternative hypotheses used in our study can be seen in Table 2.

Based on these hypotheses, we calculated the U value for each result (i.e., how often results from one type of error message are larger than from the other). In Section 4, we specify the p-value for each result (i.e., the likeliness of observing a U value this extreme

Table 2 The hypotheses for all variables.

variable	intention	hypotheses
success rate		H_0: There is no difference between the success rates of participants with OEMs and EEMs.
	correct	H_1: Participants with the EEM have a *higher* success rate than participants with the OEM.
	incorrect	H_1: Participants with the EEM have a *lower* success rate than participants with the OEM.
timing		H_0: There is no difference in the time taken to resolve the error between participants with OEMs and EEMs.
	correct	H_1: Participants with the EEM take *less* time to resolve the error than participants with the OEM.
	incorrect	H_1: Participants with the EEM take *more* time to resolve the error than participants with the OEM.
perception		H_0: There is no difference between the perceived helpfulness of OEMs and EEMs.
	correct	H_1: EEMs are rated as *more helpful* than OEMs.
	incorrect	H_1: EEMs are rated as *more misleading* than OEMs.

if there was no difference between the two distributions). 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 [15]. We report the actual p-values 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 [2].

4 Results & Implications

After cleaning our data, we recorded a total of 56 participants in our study (divided 32 / 24 over the variants). Each participant completed an average of 7 assignments (with a median of 10), for a total of 412 submissions (divided 228 / 184 over the variants). This meant that each assignment received between 37 and 55 responses with a mean of 41 and a median of 38 (original and enhanced combined). The data and the p-values from the Mann-Whitney U analysis (see Section 3.5) are summarised in Table 3. We use these to answer our research questions below. The response data as well as the assignments used in the study are available under an MIT Licence in our public data repository [26].

RQ1: How do hints in ITP error messages affect the ability of new users to fix errors?

Overall, there is strong evidence that EEMs with *correct* hints *can* improve both the likelihood that new users are able to resolve an error (**RQ1.1**) and the time taken to resolve it (**RQ1.2**). We see from Table 3 that the Confusable EEM in particular had significant influence over the participants' ability to fix the errors, and that for those that succeeded at the Whitespace assignment, the EEM significantly improved their time taken. Conversely, we see that there is no convincing evidence showing that *incorrect* hints lower the participants' success rate or that they significantly increase the time taken to find the error.

RQ2: How do new users perceive the helpfulness of hints in ITP error messages?

Table 3 A summary of our dataset as well as the results of the Mann-Whitney U test (✓ = statistically significant; × = not statistically significant; # = number of submissions considered; VM = very misleading; M = misleading; N = neutral; H = helpful; VH = very helpful; ϵ = no rating given; $\wedge$ = higher than other message type; $\vee$ = lower than other message type).

		success rate			timing				perception						
		rate		p-value	#	mean (s)	median (s)	p-value	VM	M	N	H	VH	ϵ	p-value
correct		H_1: SO(u) < SE(u)			H_1: SO(u) > SE(u)				H_1: SO(u) < SE(u)						
WHITESPACE	OEM	31 / 39	79%	0.071	29	42 $\wedge$	26 $\wedge$	0.047	5	9	8	10	5	1	<0.001
	EEM	33 / 36	92%	×	30	29 $\vee$	22 $\vee$	✓	0	0	2	3	30	1	✓
CONFUSABLE	OEM	15 / 42	36%	<0.001	11	98 $\wedge$	107 $\wedge$	0.013	6	8	9	6	5	8	<0.001
	EEM	32 / 38	84%	✓	31	60 $\vee$	54 $\vee$	✓	3	1	2	5	25	2	✓
TOOFEWARGS	OEM	30 / 44	68%	0.369	21	77 $\vee$	75 $\vee$	0.822	2	6	16	17	1	2	<0.001
	EEM	35 / 49	71%	×	31	102 $\wedge$	77 $\wedge$	×	1	4	3	26	12	3	✓
overall	OEM	76 / 125	61%	<0.001	61	64 $\vee$	50 $\vee$	0.338	13	23	33	33	11	11	<0.001
	EEM	100 / 123	81%	✓	92	64 $\vee$	51 $\wedge$	×	4	5	7	34	67	6	✓
incorrect		H_1: SO(u) > SE(u)			H_1: SO(u) < SE(u)				H_1: SO(u) > SE(u)						
WHITESPACE	OEM	26 / 41	63%	0.535	24	63 $\vee$	46 $\vee$	0.078	3	3	15	12	5	3	0.002
	EEM	27 / 42	64%	×	27	76 $\wedge$	68 $\wedge$	×	8	15	6	8	3	2	✓
TOOFEWARGS	OEM	24 / 40	60%	0.364	21	104 $\vee$	95 $\vee$	0.288	2	7	16	11	2	1	0.647
	EEM	23 / 41	56%	×	17	131 $\wedge$	109 $\wedge$	×	0	9	13	11	2	6	×
overall	OEM	50 / 81	62%	0.424	45	82 $\vee$	74 $\vee$	0.230	5	10	31	23	7	4	0.024
	EEM	50 / 83	60%	×	44	97 $\wedge$	77 $\wedge$	×	8	24	19	19	5	8	✓
		rate		p-value	#	mean (s)	median (s)	p-value	VM	M	N	H	VH	ϵ	p-value
		success rate			timing				perception						

Our data provides strong evidence for H_1 for the perception of both error message intentions: *correct* EEMs are rated as *more helpful* than OEMs and *incorrect* EEMs are rated as *more misleading* than OEMs. The only outlier seems to be the *incorrect* intention of TooFewArgs where, on average, participants actually rated the EEM *higher* than the OEM (though not significantly). We suspect that this is because by telling a participant that the number of arguments provided does not match what a function requires, they were able to at least pinpoint the problem area, even though they did not necessarily know how to fix it. For example, consider the (*incorrect*) TooFewArgs#C assignment:

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

(lines with "+" contain the enhancement)

```
  solution.agda:18.37-47: error: [UnequalTerms]
      (List _A_20 →_B_21) !=< List A
  when checking that the inferred type of an application
      List _A_20 →_B_21
  matches the expected type
      List A
+ (did you supply too few arguments to foldr ?)
```

The solution was to replace `foldr` with `intersperse`:

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

By pointing to `foldr`, participants knew they had to fix *something* about that particular function call, even though they might not have known what. On the other hand, the success rate for this particular debugging assignment actually *decreased* with the EEM, suggesting that the participants might have been misled by the error message due to unjustified trust in the hint being correct.

RQ: How do hints in ITP error messages affect their usability for new users?

From the results, we can conclude that adding hints to ITP error messages does improve their usability for new users. The actual influence they can have on the users' ability to fix an error seems to depend on the type of error that is being enhanced, meaning that it might be worthwhile to pinpoint which types of errors would use hints effectively. However, it seems that there is no practical danger in adding hints even if they might not always be displayed at the correct moment. This leads us to believe that hints in ITP error messages are a safe and viable step on the path towards making ITPs more widely accessible.

What we found very interesting was just how important the hints seemed to be for the perceived helpfulness of the error messages. Despite *incorrect* hints having no significant influence on the success rate and timing, we saw that the participants recognised how misleading the error messages were. Even more strongly, our results show that *correct* hints clearly increased the perceived helpfulness of *all* error messages we studied. We feel that this is an important indicator that even if a small usability change might not have a big impact on how people use an ITP, it might still greatly influence their developer experience with it. Hopefully, hints in the correct error messages could help improve the "bad usability" image ITPs currently seem to be suffering from.

5 Limitations & Threats to Validity

In order to contextualise the findings, we consider the limitations and threats that could have impacted the representativeness of the results:

- **Recruitment:** Since the participants were recruited during a lecture, this limits the sample population to a very specific group of people (i.e., bachelor-level, English-speaking, mostly male, computer science students with good attendance). This may not be representative of the general population of new Agda users, or of the users Agda might wish to attract.
- **Multiple Lecture Topics:** The code snippets in the study were based on different exercises covered in the course. However, the distribution of the topics over the error messages was not even, as they were randomly assigned. This could have made comparisons between different groups of error messages less reliable. For example, the ConfusableArgs snippets covered much more recent material than the TooFewArgs ones.
- **Delayed Exercise Shuffling:** To decrease fatigue effects and minimise collaboration, we shuffled the code snippets for each participant. However, during the study, the shuffling only activated after the participant opened the first assignment (TooFewArgs`#A`), meaning that every participant received this assignment, skewing response rates.
- **User Interface:** The user interface of the code editor was not representative of the standard Agda user interface. This was in part due to it using Haskell syntax highlighting, which gave different information to the user than a local Agda editor would[7]. Further, the environment did not provide functionality for entering non-ASCII Unicode, or display column information for where the cursor was in the code. All of these discrepancies could have skewed the results, as user interfaces are known to have a large impact on how programmers write code [17].
- **The Code Snippets:** The participants were asked to fix errors in code they did not write, without knowing the intent or thought process behind the code. This is likely not a standard situation for an Agda user to be in.
- **Abandon Rate:** With the software used to distribute the user study, we had no way of differentiating people who opened an exercise without intending to solve it, and those who attempted to resolve the error without saving their changes to the code. Thus, the real success rate might be higher than identified, despite the data cleaning we applied.
- **EEM Frequency:** Our study only looked at the effects of EEMs appearing *too frequently*, but not at the effects of them not appearing frequently *enough* (i.e., in situations where the EEM might be expected but does not appear).
- **Error Fatigue:** Since we were only able to study the effect of EEMs at a single point in time, we did not have a chance to evaluate how a series of *incorrect* hints affects the users' attention to them. If *incorrect* EEMs are shown too often, users might stop trusting them and learn to ignore the hints completely.

6 Related Work

Even though interactive theorem provers are known to have usability issues, research focused on addressing them is currently still scarce. Nevertheless, we highlight the most relevant work on ITP usability in this section, and discuss research on enhancing error messages for programming languages in general.

[7] For example, in a normal Agda editor, syntax highlighting is not shown if type checking fails, which is a known complaint among users [22].

Usability of Interactive Theorem Provers

A pioneer in this field, Kadoda studied the usability of a variety of theorem provers and developed a novel editor for formalising specifications [24]. In a later work, she collaborated with Stone and Diaper on using the "Cognitive Dimensions" framework [18] to evaluate educational proof editors. They found that having meaningful error messages has a high effect on the learnability of such systems [25].

A more recent line of work by Beckert and Grebing studied the usability of the KeY System, "a platform of software analysis tools for sequential Java" [3]. They found proof guidance to be important for usability, including good feedback when a proof attempt fails [10]. A later study with focus groups by Beckert et al. compared the KeY System to Isabelle [31] and found that unintuitive errors resulting from unexpected type inference are a drawback of Isabelle [11].

Most recently, Juhošová et al. investigated the obstacles that new users face when learning to use Agda specifically [22]. They found that confusing error messages are one of the most significant barriers for new users, among a variety of ecosystem obstacles.

Enhancing Error Messages

In 2019, Becker et al. conveniently conducted a literature review on enhancing error messages for programming languages in general [7]. It is important to note that these studies focus on programming novices, whereas we studied newcomers to Agda who already have programming experience in other languages.

Among the studies discussed in the literature review, the results are somewhat mixed, suggesting small or hard-to-measure benefits. Becker found that enhancing Java's error messages reduces the number of errors made by novices and improves user experience [5, 8] and later confirmed these results with a control study [6]. Pettit et al., on the other hand, found no significant effect when testing enhanced error messages for C++ [33]. In a subsequent quiz-based experiment similar to our own, Becker et al. found enhanced error messages to be effective, but the signal may be weak, which could explain the conflicting results [9].

Similar to our findings, previous work reported on students often requesting and appreciating enhanced error messages with hints or proposed solutions, but did not study their practical influence on, e.g., the success rate. Marceau et al. even suggested that error messages should not propose solutions, as they might lead novices in the wrong directions [28]. The main empirical study on enhanced error messages with hints was by Thieselton and Treude, who developed a tool for enhancing Python error messages by proposing solutions found on Stack Overflow [39]. Most participants in their study found the proposed solutions helpful, which correlates with our own observations.

7 Conclusion

Based on our between-subject user study, we recommend adding a hint to ITP error messages where deemed to potentially improve usability. Hints can improve the helpfulness of the error message as perceived by the new user and provide practical help in terms of successfully fixing the error and time taken to do so. Additionally, they do not seem to have a practical influence on these variables when hinting at an incorrect solution. In the future, we recommend research into the following directions:

- identifying which ITP error messages are most likely to benefit from hints,
- studying the influence of hints on more experience users, and
- experimenting with other types of ITP error message enhancements.

References

1 Warning against Unicode Confusables - Internals & Design, 2024. URL: https://discourse.julialang.org/t/warning-against-unicode-confusables/108734.

2 Herman Aguinis, Matt Vassar, and Cole Wayant. On Reporting and Interpreting Statistical Significance and p Values in Medical Research. *BMJ Evidence-Based Medicine*, 26(2):39–42, April 2021. doi:10.1136/bmjebm-2019-111264.

3 Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In *Verified Software: Theories, Tools and Experiments*, pages 55–71. Springer, 2014. doi:10.1007/978-3-319-12154-3_4.

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

5 Brett Becker. An Exploration Of The Effects Of Enhanced Compiler Error Messages For Computer Programming Novices. Master's thesis, Technological University Dublin, 2015. URL: https://arrow.tudublin.ie/ltcdis/35.

6 Brett A. Becker. An Effective Approach to Enhancing Compiler Error Messages. In *Technical Symposium on Computing Science Education*, pages 126–131. ACM, 2016. doi:10.1145/2839509.2844584.

7 Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, pages 177–210. Association for Computing Machinery, 2019. doi:10.1145/3344429.3372508.

8 Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175, 2016. doi:10.1080/08993408.2016.1225464.

9 Brett A. Becker, Kyle Goslin, and Graham Glanville. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Technical Symposium on Computer Science Education*, SIGCSE '18, pages 640–645. Association for Computing Machinery, 2018. doi:10.1145/3159450.3159461.

10 Bernhard Beckert and S. Grebing. Evaluating the usability of interactive verification systems. *CEUR Workshop*, 2012.

11 Bernhard Beckert, Sarah Grebing, and Florian Böhl. A Usability Evaluation of Interactive Theorem Provers Using Focus Groups. In *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, September 2014. doi:10.1007/978-3-319-15201-1_1.

12 Ben Hamilton. Data.Text.ICU.Spoof, 2015. URL: https://hackage.haskell.org/package/text-icu-0.8.0.5/docs/Data-Text-ICU-Spoof.html.

13 Lawrence Chonavel. Pull Request #7094: Move Unicode Keybindings into JSON File by Lawcho, 2024. URL: https://github.com/agda/agda/pull/7094.

14 Leonardo de Moura and Ullrich, Sebastian. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction (CADE 28)*, pages 625–635. Springer-Verlag, 2021. doi:10.1007/978-3-319-21401-6_26.

15 R. A. Fisher. Statistical Methods for Research Workers. In Samuel Kotz and Norman L. Johnson, editors, *Breakthroughs in Statistics: Methodology and Distribution*, pages 66–70. Springer, New York, NY, 1992. doi:10.1007/978-1-4612-4380-9_6.

16 Frederik Hanghøj Iversen. Issue #2898: Suggest Replacements for Various Unicode Characters, 2018. URL: https://github.com/agda/agda/issues/2898.

17 Gavin Gray, Will Crichton, and Shriram Krishnamurthi. An Interactive Debugger for Rust Trait Errors. *Artifact for An Interactive Debugger for Rust Trait Errors*, 9(PLDI):199:1293–199:1314, 2025. doi:10.1145/3729302.

18 T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. doi:10.1006/jvlc.1996.0009.

19 Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028. ACM, 2010. doi:10.1145/1753326.1753478.

20 Heman Gandhi. How Not to Learn Agda. URL: https://hemangandhi.github.io/blog-posts/agda.html.

21 Josip Medved. Unicode Code Point, 2025. URL: https://marketplace.visualstudio.com/items?itemName=medo64.code-point.

22 Sára Juhošová, Andy Zaidman, and Jesper Cockx. Pinpointing the Learning Obstacles of an Interactive Theorem Prover. In *International Conference on Program Comprehension (ICPC)*, pages 159–170. IEEE, 2025. doi:10.1109/ICPC66645.2025.00024.

23 Sára Juhošová, Andy Zaidman, and Jesper Cockx. The Way of Types: A Report on Developer Experience with Type-Driven Development. In *International Conference on Program Comprehension (ICPC)*. ACM, 2026. doi:10.1145/3794763.3794812.

24 Gada F. Kadoda. *Formal Software Development Tools: An Investigation into Usability*. PhD Thesis, Loughborough University, 1997. URL: https://hdl.handle.net/2134/31907.

25 Gada F. Kadoda, Roger G. Stone, and Dan Diaper. Desirable features of educational theorem provers - a cognitive dimensions viewpoint. In *Annual Workshop of the Psychology of Programming Interest Group*, 1999.

26 Maria Khakimova, Sára Juhošová, and Jesper Cockx. Dataset for: "Enhancing Interactive Theorem Prover Error Messages with Hints", 2026. doi:10.4121/10425039-1e9c-46d2-80c9-a2805c1b7e62.

27 Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, San Francisco, CA, USA, second edition, 2022.

28 Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind Your Language: On Novices' Interactions with Error Messages. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18. ACM, 2011. doi:10.1145/2048237.2048241.

29 Michael Nahas. Issue #5629: Unicode Identifier Visually Similar to ASCII One (Source of Confusion/Obfuscation), 2021. URL: https://github.com/agda/agda/issues/5629.

30 Nadim Nachar. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1), 2008. doi:10.20982/tqmp.04.1.p013.

31 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

32 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD Thesis, Chalmers University of Technology, Göteborg, Sweden, 2007.

33 Raymond S. Pettit, John Homer, and Roger Gee. Do Enhanced Compiler Error Messages Help Students? Results Inconclusive. In *Technical Symposium on Computer Science Education*, SIGCSE '17, pages 465–470. ACM, 2017. doi:10.1145/3017680.3017768.

34 Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students? In *Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 182–187. ACM, 2017. doi:10.1145/3059009.3059058.

35 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

36 The Agda Development Team. Emacs Mode, 2025. URL: https://agda.readthedocs.io/en/v2.8.0/tools/emacs-mode.html.

37 The Agda Development Team. Mixfix Operators, 2025. URL: https://agda.readthedocs.io/en/v2.8.0/language/mixfix-operators.html.

38 The Coq Development Team. The Coq Proof Assistant, September 2024. URL: https://zenodo.org/records/14542673.

39 Emillie Thiselton and Christoph Treude. Enhancing Python Compiler Error Messages via Stack. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. ISSN: 1949-3789. doi:10.1109/ESEM.2019.8870155.

40 V. Javier Traver. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction*, 2010(1):602570, 2010. doi:10.1155/2010/602570.

41 Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht, editors, *Advances in Web-Based Learning*, pages 228–239. Springer, 2012. doi:10.1007/978-3-642-33642-3_25.

42 zeithaste. Unicode Code Point of Current Character, 2024. URL: https://marketplace.visualstudio.com/items?itemName=zeithaste.cursorCharCode.