How Novices Perceive Interactive Theorem Provers

SÁRA JUHOŠOVÁ, Delft University of Technology, The Netherlands

1 INTRODUCTION

In a world where software is present in all aspects of our lives and where small code bugs can have catastrophic consequences [12], program verification becomes increasingly more essential. There are many known techniques for such verification, including model-checking [3], abstract interpretation [5], and interactive theorem provers (ITPs) - the focus of our study. ITPs allow human users and computers to "work together interactively to produce a formal proof" [9, p. 135], essentially allowing users to reason about their programs and use the computer to verify their specifications. Examples of ITPs include Agda [15], Coq [4], and Lean [13].

While ITPs are powerful tools for ensuring program correctness, they have trouble spreading into commercial software development [6, 8, 11]. One possible explanation for the lack of industrial adoption is their poor usability. Despite an awareness of this problem within the field, user studies and user-oriented design are rare and ITPs remain inaccessible to a wide range of practitioners.

Our long-term goal is to identify the most significant aspects of ITPs to improve with respect to usability, and outline design guidelines that ITP maintainers can follow. Here, we start at the beginning: we examine the obstacles novices face when learning to use an interactive theorem prover and discuss the implications and possibilities for improving the accessibility of ITPs.

2 STUDY SETUP

The participants in this study were bachelor students taking a Functional Programming course at the TU Delft. The data was collected through an online survey, distributed to the students after a two-week introduction to Agda and open for a period of three weeks. The survey had an open-ended question with five short free-text fields, where they could use their own words to describe the obstacles they had encountered. We did not want to restrict the answers to preconceived obstacles and risk missing the opportunity to discover something unexpected.

During the course, the students attended live lectures about Agda and had multiple exercises to practice on. They were encouraged to use the `agda-mode` extension in Visual Studio Code (VSC). Of the 35 participating students (P1-35), only one indicated that their knowledge of Agda (or another ITP) was more advanced than what was taught in the course. We therefore identify the group as "novices" in interactive theorem proving.

We analysed the data by performing two separate iterations of coding, i.e. "systematically categorizing excerpts in [our] data in order to find themes and patterns" [1]. In the first iteration, we used descriptive coding [14, p. 55 - 69] to help us identify types of obstacles occurring in our data. This process was done inductively, i.e. without preconceived codes.

In the second iteration, we used the identified obstacle types from the first iteration to reclassify each submitted obstacle. We found that the data contained more subtleties than a simple descriptive code could capture, and decided to add sub-codes for each submission. These were coded on a more granular level and added interpretation to the description.

For example, consider the following two submitted obstacles:

- "The Agda plugin in my [VSC] was often crashing and I had to restart it." [P18]
- "Syntax highlighting completely disappears if there is some mistake in the code which makes it harder to find the mistake." [P15]

Author's address: Sára Juhošová, Delft University of Technology, Delft, The Netherlands, S.Juhosova@tudelft.nl.

Both relate to the quality of developer tools provided for Agda, thus receiving the code `tooling`. However, while one talks about unintended behaviour (crashing), the other relates to the design of the tool (syntax highlighting only appears on successfully compiled files). By capturing these nuances in the sub-codes, we were able to better characterise each obstacle.

When the coding was done, we drew diagrams [2, p. 218 - 221], using sub-codes to help us understand the relationships between the identified obstacle types. These diagrams helped us identify related categories of obstacles, presented in Section 3. They also prompted the writing of memos, notes of "ideas about codes and their relationships as they strike the analyst" [7], which form the basis of our interpretation in Section 4.

3 IDENTIFIED OBSTACLES

During the analysis, we identified five categories of obstacles related to Agda's theory (T), implementation (I), and applicability in the real world (A). We outline them below.

Unfamiliar Concepts (T). Agda is very different from most programming languages that bachelor students are used to. Using dependent types is "not intuitive" [P13] and the idea that the "magic happens during type checking instead of execution [takes time] to wrap [their] head around" [P31].

Complex Theory (T). A significant amount of understanding of the underlying theory is needed to reason about code in Agda. "The way you need to think about your program [is] much more abstract" [P28] and effective use of Agda's powerful features requires familiarity with their underlying principles (e.g. recognising when Agda is and is not able to unify two terms via `refl`).

"Weird" Design (I). The most mentioned type of obstacle (mentioned by 23 participants) was Agda's "weird" [P6] design. Examples of this include:

- the use of Unicode characters which "[raise the] barrier of entry" [P17],
- an enforcement of spacing rules that are not standard in other programming languages,
- error messages that require theoretical knowledge and experience to be helpful,
- and abstractions which require an understanding of the implementation.

Inadequate Ecosystem (I). The ecosystem that supports program-writing in Agda is

- *incomplete* ("[there is a] lack of online courses/resources on Agda" [P34]),
- *buggy* ("the installation of Agda is a horrible experience" [P4]),
- *inconvenient* ("syntax highlighting [is] not updating automatically and not highlighting anything on invalid syntax" [P29]),
- and *not accessible to novices* ("the Agda docs have a lot of material on super crazy stuff, but very little on the detailed semantics of the basic language" [P5]).

Eighteen of all participants mentioned issues with the ecosystem, including crashing editor plugins, missing documentation, or under-specified interactive features (e.g. automatic proof search).

Perceived Irrelevance (A). Being taught to use Agda as a proof assistant, students find it "hard to imagine writing software with Agda" [P28]. They are used to creating and testing software that interacts with humans in the real world, which requires IO as well as integration with other software development tools. Having experienced neither, they are left feeling that "[Agda] might be a bit too theoretical" [P23] and struggle to find its relevance.

4 INTERPRETATION & IMPLICATIONS

There are three interesting observations we can make based on the identified obstacles.

First, many of these obstacles are a result of the high coupling between Agda's underlying theory and its design. "The relative complexity of the theories underlying [ITPs] makes them inaccessible to

a wide range of software engineers who are not experienced mathematicians" [10, p. 1]. Developers need a sturdy grasp of what is going on under the hood to write programs, understand errors, and make use of Agda's interactive features and automation.

Second, despite the amount of unfamiliar concepts novices need to get used to, Agda's ecosystem has very little supporting infrastructure for them. The syntax is unusual, the use cases differ from more common programming languages, and the documentation is not complete and accessible enough to bridge those differences. Similarly, Agda's standard libraries are difficult to work with, their design aimed at experienced ITP programmers. This, coupled with a novice's difficulty to imagine where Agda could be applied, causes a frustrating onboarding experience.

Thirdly, Agda's design choices (e.g. interactive commands or Unicode syntax) make it dependent on a custom development environment. At the same time, the existing one consists of tools riddled with bugs and confusing information. There seems to be a pattern of well-intended features (such as installation through an editor plugin) which do not reach the quality necessary to be practical.

Error messages, which were mentioned by 12 participants, are a demonstrative example of our interpretation. Consider the following piece of code:

```
1 | swap : a × b → b × a
2 | swap (a, b) = b , a
```

Upon loading this code in VSC, the `agda-mode` extension will report the following error:

```
Could not parse the left-hand side swap (a, b)
Problematic expression: (a, b)
Operators used in the grammar: None
when scope checking the left-hand side swap (a, b) in the definition of swap
```

This behaviour causes several difficulties for a novice: (1) the message contains obscure information that needs to be filtered out, (2) an online search using the error message yields no relevant results, and (3) all syntax highlighting will be removed, making the error even more difficult to spot. That is a lot of cognitive overhead for an error as simple as "a space is missing (Ln 2, Col 8)".

Our results offer a clear and simple point of improvement: provide more accessible and sturdy infrastructure. To make ITPs more accessible to a wider range of practitioners, we need to provide them with the support to build a foundation and master the basics. This can be as simple as completing the documentation, maintaining the development environment, and applying user feedback on ITP design. We see these tasks as an opportunity to broaden the community and strengthen the field by investing more into ITP developer experience.

5 LIMITATIONS & FUTURE WORK

This study was conducted on a rather homogenous set of students, learning the basics of Agda in a short span of time. Their answers might have been influenced by the lecturing style and content, the formulation of the survey questions, as well as the idiosyncrasies of Agda compared to other ITPs. In order to overcome these limitations, we hope to conduct more research into the usability of ITPs, focusing on a more varied audience and a wider selection of ITPs.

Additionally, while this study highlights the obstacles that one might face when learning to use Agda, it does not provide solutions for the technical challenges. The underlying theory does not only affect how ITPs are used, but also how they are designed. Research into creating "intuitive" features that fit within the paradigm of ITPs would be an interesting and challenging endeavour.

REFERENCES

[1] [n. d.]. Essential Guide to Coding Qualitative Data. https://delvetool.com/guide

[2] Kathy Charmaz. 2014. *Constructing Grounded Theory* (2 ed.). https://uk.sagepub.com/en-gb/eur/constructing-grounded-theory/book235960

[3] Edmund M. Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science (Lecture Notes in Computer Science)*, S. Ramesh and G. Sivakumar (Eds.). Springer, 54–56. https://doi.org/10.1007/BFb0058022

[4] Coq Team. [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '77)*. ACM, 238–252. https://doi.org/10.1145/512950.512973

[6] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. 2019. A framework for improving error messages in dependently-typed languages. *Open Computer Science* 9 (Jan. 2019), 1–32. https://doi.org/10.1515/comp-2019-0001

[7] Barney G. Glaser. 1978. *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Press, Mill Valley, CA.

[8] Sarah Grebing and Mattias Ulbrich. 2020. Usability Recommendations for User Guidance in Deductive Program Verification. In *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich (Eds.). Springer International Publishing, 261–284. https://doi.org/10.1007/978-3-030-64354-6_11

[9] John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of Interactive Theorem Proving. In *Handbook of the History of Logic*, Jörg H. Siekmann (Ed.). Computational Logic, Vol. 9. North-Holland, 135–214. https://www.sciencedirect.com/science/article/pii/B9780444516244500046

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

[11] Philipp Kant, Kevin Hammond, Duncan Coutts, James Chapman, Nicholas Clarke, Jared Corduan, Neil Davies, Javier Díaz, Matthias Güdemann, Wolfgang Jeltsch, Marcin Szamotulski, and Polina Vinogradova. 2020. Flexible Formality Practical Experience with Agile Formal Methods. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, 94–120. https://doi.org/10.1007/978-3-030-57761-2_5

[12] Amy J. Ko, Bryan Dosono, and Neeraja Duriseti. 2014. Thirty years of software problems in the news. In *Proc. of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2014)*. ACM, 32–39. https://doi.org/10.1145/2593702.2593719

[13] Lean FRO. [n. d.]. Programming Language and Theorem Prover — Lean. https://lean-lang.org/

[14] Matthew B.. Miles and Michael Huberman. 1994. *Qualitative data analysis: an expanded sourcebook* (2nd ed ed.). Sage, Thousand Oaks, CA.

[15] The Agda Development Team. 2024. Agda. https://wiki.portal.chalmers.se/agda/pmwiki.php