Pinpointing the Learning Obstacles of an Interactive Theorem Prover

Sára Juhošová
S.Juhosova@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
A.E.Zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Jesper Cockx
J.G.H.Cockx@tudelft.nl
Delft University of Technology
Delft, The Netherlands

***Abstract*—Interactive theorem provers (ITPs) are programming languages which allow users to reason about and verify their programs. Although they promise strong correctness guarantees and expressive type annotations which can act as code summaries, they tend to have a steep learning curve and poor usability. Unfortunately, there is only a vague understanding of the underlying causes for these problems within the research community. To pinpoint the exact usability bottlenecks of ITPs, we conducted an online survey among 41 computer science bachelor students, asking them to reflect on the experience of learning to use the Agda ITP and to list the obstacles they faced during the process. Qualitative analysis of the responses revealed confusion among the participants about the role of ITPs within software development processes as well as design choices and tool deficiencies which do not provide an adequate level of support to ITP users. To make ITPs more accessible to new users, we recommend that ITP designers look beyond the language itself and also consider its wider contexts of tooling, developer environments, and larger software development processes.**

***Index Terms*—interactive theorem provers, learning obstacles, Agda**

I. Introduction

In a world where technology is present in all aspects of our lives and where each program bug can have catastrophic consequences [1], verifying the correctness of the software we produce is an essential part of its development. Current methods for such verification can be split into two categories:

1) ones which help us *detect* errors in our code, and
2) ones which *prevent* certain types of errors in our code.

Commonly used error-detection methods, such as code review [2], [3] and testing [4], [5], are well-integrated into software development processes, with plenty of frameworks and supporting material to help developer teams apply them. However, they have one big disadvantage: they are only able "to show the presence of bugs, but never to show their absence"[1]. This means that error-detection tools are not sufficient to make sure that our software works correctly in all cases.

Error-prevention tools, on the other hand, offer stronger guarantees about our programs and can potentially provide mathematical proof of their correctness with respect to some specification. *Static type systems* fall into this category. Defined by Benjamin Pierce as a "method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute" [6, p. 1], they ensure that it is impossible to write programs such as one which tries to retrieve the i^{th} element of a `boolean` value. Type systems are a notable field of study within the programming language community and are built into many mainstream programming languages, including Java and C++. They come in various advanced forms, like with Hindley-Milner type inference as used in Haskell [7] or as substructural types as used in Rust [8].

Dependent types are an example of a powerful type system, going beyond the expressivity of those in Java or Haskell. They might, for example, prevent you from trying to retrieve the 6^{th} element from a `list` which only contains five elements. This is done by allowing types to depend on terms, two worlds which are usually kept separate [9]. The resulting type signatures are so expressive that they can act as "free" and up-to-date program comprehension aids — similar to how assertions can "help express the purpose of a [function] without reference to its implementation" [10, p. 46] in programs written using design by contract principles. This is a big benefit, since we know that "missing or outdated comments can substantially impair the development process" [11]. Additionally, most programming languages with a dependent type system can be used as interactive theorem provers[2] (ITPs), allowing human users and computers to "work together interactively to produce a formal proof" [12, p. 135]. Examples of ITPs include Agda [9], Coq [13], and Lean [14].

Unfortunately, existing languages with dependent types provide their correctness guarantees and expressive type signatures at a cost: they are difficult to use. The consensus within the broader ITP community is that the learning curve is very steep and that the development and maintenance process for produced code is expensive. Despite this awareness, user studies and user-oriented design are exceedingly rare in the field of programming languages [15], and we lack a clear idea about what the exact usability bottlenecks of ITPs are. Consequently, we do not know where to start and what to prioritise when improving the accessibility of these languages.

In this study, we take a step towards closing this knowledge gap and investigate the following research question: *how can*

[1]Famously stated by Edsger W. Dijkstra.

[2]As a consequence, we mean "dependently-typed language used as an ITP" whenever we mention ITP in the remainder of this paper.

we make interactive theorem provers more accessible to new users? Based on a survey among 41 bachelor students who recently learned to use Agda[3] about their experience and the obstacles they faced during the process, we were able to make the following contributions:

- A hypothesis about how new users perceive the role of interactive theorem provers within software development.
- A list of obstacle types that new users encounter when learning to use Agda, grouped based on whether they are related to Agda's theory, its implementation, or its perceived role within software engineering. This knowledge can be used to fix the identified obstacles in Agda and avoid them in other ITPs.
- An overview of how frequently these obstacle types occur and how severe new users consider them to be. This knowledge can help ITP designers prioritise the most pressing issues.

We conclude that there are two actions designers of any ITP can take in order to mitigate the learning obstacles found in Agda: integrating the ITP into software processes and ecosystems[4] and providing its users with a more robust and accessible infrastructure.

II. An Introduction to ITPs

Currently, the core of most interactive theorem provers is *purely functional programming*. This means that functions are treated as first-class citizens and that they do not have any side effects [16, p. 13]. Additionally, all functions written in an ITP must be *total*, guaranteeing that a result will be produced for every valid input in finite time and that no runtime exceptions will be thrown [16, p. 16]. Listing 1 showcases the definition of `lookup`, an example of a *pure, total function* in Agda, which takes an index and a list, and *maybe* returns the element at that position (depending on whether the position exists in the provided list). The definition works in the following way:

`Nat`: First, we define natural numbers as either `zero` or the successor (`suc`) of another natural number. The number 2 would be represented as `suc (suc zero)`.

`Maybe A`: Next, we define the `Maybe` data type, parametrised by the type variable `A`. This data type can be constructed using either `nothing` or `just x`. In the latter case, `x` is a value of type `A`, wrapped in the `just` constructor.

`List A`: Finally, we define `List` as either an empty list (`[]`) or an element of type `A` prepended to another `List A` (denoted as `x :: xs`). For example, `zero :: (suc zero) :: []` is a list of natural numbers with two elements (`[0, 1]`).

The `lookup` function matches on the two types of list constructors and defines the behaviour for each of those cases:

1) In case the list is empty, we ignore the index and return `nothing` (any index will be out of bounds).
2) In case the list contains at least one element, we can further match on the natural number that represents the index we want to retrieve from:
   a) In case the index is `zero`, we return the current first element wrapped in a `just` constructor and ignore the remainder of the list.
   b) In case the index is larger than `zero`, we ignore the current first element and recursively search the remainder of the list, decreasing the index by one.

```
data Nat : Set where
    zero : Nat
    suc  : Nat → Nat

data Maybe (A : Set) : Set where
    nothing : Maybe A
    just    : A → Maybe A

data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A

lookup : Nat → List A → Maybe A
lookup _       []         = nothing
lookup zero    (x :: _)   = just x
lookup (suc i) (_ :: xs) = lookup i xs
```

Listing 1: Retrieving an element from a list in Agda

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

lookup : {n : Nat} → Nat< n → Vec A n → A
lookup ()       []
lookup zero     (x :: _)   = x
lookup (suc i) (_ :: xs) = lookup i xs
```

Listing 2: Retrieving an element from a vector in Agda

Although this implementation guarantees that an "index out of bounds" exception will never occur, it has a drawback: we will have to unwrap our element from the `Maybe` type every time we call the `lookup` function. Using dependent types, we can avoid this issue by limiting the index to natural numbers that are smaller than the length of the list. Listing 2 contains a definition of a vector which is "indexed" by its length and the same `lookup` function defined for this new data type. This function *depends on* the natural number `n` which defines both the upper bound for the index[5] and the size of the vector. Since Agda's type-checker understands that it is impossible to call this function with an out-of-bounds index, we do not have to wrap our return value in a `Maybe` in order to preserve totality.

Proofs, just like functions, are first-class citizens in ITPs. For example, consider the function `double` (defined in Listing 3) which takes a natural number and returns (1) a number that is the double of the input number, and (2) a proof that the

[3]Agda is one of the main dependently-typed ITPs and is taught at our university, giving us access to a controlled set of new users: students.

[4]Under an "ecosystem" we understand all tools and libraries that facilitate program-writing and software development in a programming language.

[5]The type "`Nat< n`" is defined as "a natural number smaller than `n`".

```
double : (n : Nat) → m is Nat and m IsEven
double zero = (zero , ZeroIsEven)
double (suc n) = case (double n) of λ where
    (m , mIsEven) →
        (suc (suc m) , Since mIsEven SucSucIsEven)
```

Listing 3: An Agda function returning a doubled natural number and a proof that the result is even

resulting number is even. We can only define this function if it is possible to construct such a proof (which is indeed the case, since all numbers multiplied by `2` are even). Both the number and the proof can be extracted from the result and used by other functions. We have thus both verified the evenness of our result and made the proof available as a building block for more complex proofs to the rest of the program.

"`m is Nat and m IsEven`" is syntactic sugar for a dependent pair in which the type of the first element is natural number and the type of the second element is the function `λ x → x IsEven` applied to the first element (it *depends on* the first element). This dependency explicitly tells the type-checker to verify that our program returns a proof of the first element's evenness — not the evenness of any random number.

An Illustration of Interactivity

What makes a theorem prover interactive is the way it allows programmers to communicate with the type-checker. In Agda, for example, you can create a "hole" (denoted by `{!  !}`) as a placeholder for an expression you do not yet know how to fill. This will make a number of interactive commands available [17]. For example, a user might want to define a `map` function over vectors which applies some function to each element in the vector. They could define it using six interactive steps.

1) Write the type signature and use a hole for the definition:

```
1 | map : (A → B) → Vec A n → Vec B n
2 | map f xs = {!   !}
```

2) Have Agda enumerate all possible cases for the input vector, resulting in two holes:

```
2 | map f [] = {!   !}
3 | map f (x :: xs) = {!   !}
```

3) Have Agda automatically fill the first hole [18] (only one option has the correct type):

```
2 | map f [] = []
```

4) Have Agda search for and apply a unique way to construct the required type:

```
3 | map f (x :: xs) = {!   !} :: {!   !}
```

5) Ask Agda for contextual information about the first hole. It requires a value of type `B`, and we have `x` of type `A` and `f` of type `A → B` available. There is only one way to combine those in the hole:

```
3 | map f (x :: xs) = (f x) :: {!   !}
```

6) Have Agda automatically fill the final hole (only one option has the correct type):

```
1 | map : (A → B) → Vec A n → Vec B n
2 | map f [] = []
3 | map f (x :: xs) = (f x) :: (map f xs)
```

A Demonstration of Difficulties

To demonstrate the difficulties a new user might face when learning to use an interactive theorem prover, we use the example of a simple type error. Such errors can prove challenging to resolve even in simpler type systems since they may "point to locations that are not the root causes of the type error, expose errors in cryptic language, or provide misleading fixing suggestions" [19]. Consider the following piece of Agda code, which takes a pair of value==uand swaps their positions:

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

Upon attempting to type-check this in VS Code, the `agda-mode` extension reports 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 poses several difficulties for a new user:

1) the message contains seemingly irrelevant information (such as "operators used in grammar"),
2) an online search using the error message yields no relevant results, and
3) all syntax highlighting will be removed from the file, making the error even more difficult to spot.

That is a lot of cognitive overhead for an error as straightforward as "there is a space missing on line 2, column 8".

While this looks like a simple example, it is a direct result of Agda's variable-naming design. The pattern matching assumes that "`a,`" should be treated as a constructor (since it would be a valid name for one) and warns that no such constructors are available for the type that is being matched. More examples of confusing error messages include

- a "Parse error", which *might* indicate a missing bracket (or anything else),
- a suggestion that "⟨" is not known, but did the user perhaps mean "⟨" (visually identical but with a different Unicode encoding), and
- a list of garbled characters representing implicit variables that Agda cannot unify without pointers to their origin.

A Peek at Previous Work

The few user-oriented studies that have been conducted on ITPs in the past focus primarily on proof-writing and proof-comprehension services. Our work investigates the new user's experience specifically and examines the obstacles associated not only with the ITP but also with its surrounding ecosystem. Additionally, as far as we are aware, no study has been conducted on dependently-typed ITPs. Section VII goes into detail about related work and summarises their findings in context of ours.

III. Study Setup

In this cross-sectional study, we asked students to evaluate the applicability of ITPs on a Likert scale and to list up to five obstacles they encountered when learning Agda. We were examining this population of students at one determined point in time [20, pp. 105–106]: just after they had a short introduction to programming in Agda and could be considered "new users". This section presents the details about the participants we recruited, the survey design we used, and the analysis we performed on the collected data. Our research was conducted with the approval of the Human Research Ethics Committee of the Delft University of Technology.

A. Context & Participants

The participants in this study were final-year computer science bachelor students taking an elective Functional Programming course at the Delft University of Technology. During the course, they learned the basics of functional programming in Haskell followed by a two-week introduction to Agda where they were taught to

- interactively develop Agda programs,
- use the Curry-Howard correspondence [6, 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. They were encouraged to use the `agda-mode` extension in the Visual Studio (VS) Code editor. We estimate that each student spent about 20 hours studying all Agda material taught in the course.

Of the 41 participating students, only two indicated that their knowledge of ITPs was more advanced than what was taught in the course. Additionally, four participants indicated that their knowledge of pure functional programming languages like Haskell was more advanced than what was taught in the course. Only one participant indicated both of the above.

Though we did not ask for any more personal or contextual information, we had a general idea of the background of this student group, since we are familiar with the structure of their bachelor programme. The following background information is relevant to this study:

- The participant population is somewhat skewed towards students who were already interested in functional programming. The Functional Programming course is one of six electives that are offered at the end of the bachelor programme. Students have to pick three of these electives.
- Most courses and projects about software engineering practices in the bachelor use object-oriented paradigms and focus on software that in some way collects, processes, or serves user-input data. The main programming language of instruction in the bachelor programme is Java.
- The students had experience with functional programming from the Concepts of Programming Languages course, where they were required to use the functional aspects of Scala to implement definitional interpreters. However, this was limited to recursive functions, pattern matching, and an introduction to algebraic data types.
- The main forms of ensuring program correctness that are taught within the bachelor programme are software testing and code review, both of which are error-detection methods. These are taught in the following three courses: OOP Project, Software Quality & Testing, and Software Engineering Methods.
- Learning to use Agda was not the first time our participants were confronted with having to reason about programs and construct proofs. They were expected (but not required) to already have passed the following courses:
  - Reasoning & Logic, in which they learned to construct direct and indirect proofs (including proof by mathematical induction), logical equivalences, and counterexamples for (in)valid arguments,
  - Algorithm Design, in which they learned to prove the correctness of algorithms, and
  - Automata, Computability and Complexity, in which they learned to prove and verify proofs of various problem and language properties.

  In all of these courses, the proofs were done on paper.

B. Survey

The data was collected through an online survey after the two-week introduction to Agda and open for a period of three weeks. Participation was voluntary, and the survey was distributed to the students using a QR code in the lecture slides and an announcement on the course page. A total of 224 students were enrolled in the Functional Programming course, meaning that about 18.5% of the course participants responded to the survey.

The first part of the survey consisted of a Likert scale matrix, asking students to evaluate the applicability of ITPs. The six statements used in this matrix are displayed in Figure 1. We opted to use a 6-point Likert scale with no neutral option, since we wanted to at least determine whether they tended towards a positive or negative opinion [21, p. 7]. Participants were informed that they were free to skip any question, so they could opt out of answering if they did not have an opinion.

The second part of the survey was inspired by the exploratory survey in Robillard and DeLine's field study of API learning obstacles [22]. It had an open-ended question with short, free-text fields in which students could use their own words to describe up to five obstacles they had encountered. We then asked them to rate each obstacle on a severity scale from 1 (minor inconvenience) to 10 (blocking). We chose to not restrict the answers to a list of predetermined obstacles, in order to retain the opportunity of discovering something unexpected.

Please rate the following statements on your evaluation of interactive theorem provers *[on the following Likert scale: `Strongly agree`, `Agree`, `Somewhat agree`, `Somewhat disagree`, `Disagree`, `Strongly disagree`]*.

- `E1` I would trust software more, knowing that it was verified using an interactive theorem prover.
- `E2` I think interactive theorem provers are a great alternative to testing (code is formally verified instead of tested).
- `E3` I think interactive theorem provers are a great complement to testing (some code is formally verified while the rest is tested).
- `E4` I expect there to be fewer bugs in software that has been formally verified using an interactive theorem prover as opposed to software that has been tested.
- `E5` I think interactive theorem provers will be useful in future projects (personal project or ones at a software engineering job).
- `E6` I think interactive theorem provers will be useful on critical components of future projects (personal project or ones at a software engineering job).

Figure 1: Survey question for evaluating ITP applicability

C. Pilot Study

To verify the design of our survey, we conducted a pilot study with two think-aloud sessions. One session was conducted with a master student who took the same course three years prior, and the other was conducted with a researcher who uses Agda on a daily basis. During both sessions, we asked the pilot participant to fill in the survey in our presence, voicing all thoughts and interpretations out loud. Based on their reactions and interpretations, we adjusted the wording and format of the survey questions.

The data obtained from the pilot was discarded and is not part of the results presented in this paper.

D. Analysis

In order to analyse the Likert scale data evaluating the applicability of ITPs, we assigned linear-scale numerical values[6] to all the responses [23]. This allowed us to easily interpret and visualise the data. The obstacle data, on the other hand, was free-text and so qualitative research techniques were required for its analysis. We analysed this data in two separate iterations of coding, "the process of closely inspecting, deeply making sense of, and inferring meaning from data and giving those meanings some labels or names" [24]. In the first iteration, we used descriptive coding [25, pp. 55–69] to help us identify types of obstacles occurring in our data. This process was done inductively, without predetermined codes.

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

For example, consider the following two entries:

- "The Agda plugin in my VS Code 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]

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

Throughout the rest of this paper, we differentiate between:

- *obstacle categories / types:* the identified categories of obstacles,
- *obstacle instances:* examples of specific obstacles within those categories, and
- *obstacle entries:* the separate entries provided by the participants which comprise the instances.

When the coding was done, we drew diagrams as suggested by Charmaz [26, pp. 218–221], using sub-codes to help us understand the relationships between the identified obstacle instances. These diagrams helped us identify related categories of obstacles, presented in Section IV-B. They also prompted the writing of memos, notes of "ideas about codes and their relationships as they strike the analyst" [27], which form the basis of our observations in Section V. Once we finished identifying the obstacle categories, we determined their magnitude using the severity ratings of their entries.

IV. Results

In this subsection, we present our results and offer potential explanations for them in the context of the participants' background (see Section III-A for more details). First, we discuss the participants' responses to the Likert scale applicability statements. These show us the general impressions that the participants had of interactive theorem provers after their experience of learning to use them. Then, we present the obstacle categories that we identified. These highlight the specific aspects of Agda that need improving with respect to accessibility for new users. Together, these results illustrate the current picture of what learning to use an existing ITP entails and show us what to focus on in order to *make interactive theorem provers more accessible to new users*.

A. Applicability Evaluation

Figure 2 displays a comprehensive overview of the responses to the ITP applicability evaluation statements. As a consequence of data cleaning, we removed the responses from one participant who had reacted with `Strongly agree` to all statements, leaving a total of 40 responses per statement. The omission did not alter our results.

In general, participants' opinions tended towards agreement: all statements received more positive than negative responses. We can make three interesting observations by comparing some responses:

1) *In general, participants agreed that interactive theorem provers can help deliver more trustworthy software.* Both

[6] From `1` for `Strongly disagree` to `6` for `Strongly agree`.

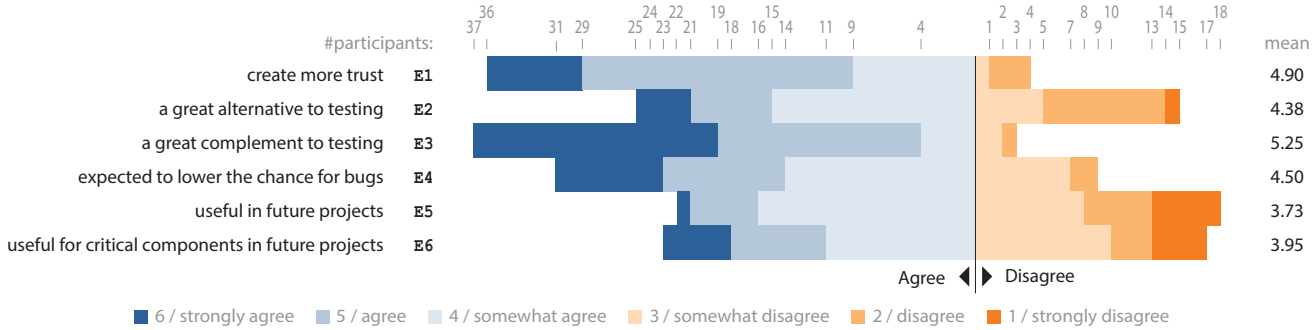

Figure 2: Overview of participants' responses to the applicability evaluation statements

the statement about "having higher trust in software if an ITP was used" (`E1`) and "expecting software to have fewer bugs if an ITP was used" (`E4`) scored high on average, and neither had any `Strongly disagree` responses. While this shows that participants believed ITPs are capable of delivering more trustworthy software, it does not address how that fits into their views of software engineering.

2) *Participants viewed interactive theorem provers as a supplementary rather than standalone tool.* This was clear when compared to their opinions about testing, a technique they are familiar with from previous courses. While responses to ITPs being "a great *alternative* to testing" (`E2`) are fairly neutral, we see an increase of almost one Likert scale point in the responses to ITPs being "a great *complement* to testing" (`E3`). Even more interestingly, the most popular response to the latter was `Strongly agree`, and only three people responded with a negative opinion. We hypothesise that ITPs are viewed as being a great *additional* tool that can *increase* trustworthiness, but that it is not necessarily ideal as the only guarantor.
3) *Participants were unsure about the usefulness of interactive theorem provers in their future projects.* "ITPs will be useful in future projects" (`E5`) and "ITPs will be useful for critical components in future projects" (`E6`) scored the lowest of all statements. In both cases, a weak `Somewhat agree` was the most popular response and there is a surprisingly small difference between the responses to the two statements. This could be because participants see potential applications for ITPs, but do not expect to use them in their own future work — regardless of whether that work is on critical components or not.

B. Identified Obstacles

35 of the 41 participating students (P1–35) answered the open question about obstacles they faced when learning to use Agda. Only one of those was a participant who indicated they had more knowledge of ITPs than was taught in the course. Of the 105 obstacles entries we received from these participants, six entries were deemed invalid:

	obstacle	*#entries*	*mean*	*mode*
(1)	unfamiliar concepts	9	5.00	5
(2)	complex theory	12	4.58	4
(3)	"weird" design	27	4.85	1 / 2
(4)	inadequate ecosystem	45	5.73	8
(5)	perceived irrelevance	6	4.83	7

Table I: Severity ratings per obstacle category

- three referred to the course organisation and design instead of to Agda in general,
- two were complaints about what students incorrectly assumed were missing features (they were not introduced in the course), and
- one simply stated that there were no more obstacles to mention.

Based on the 99 valid entries, we identified five categories of obstacles related to the theory, implementation, and practical applications of Agda:

theory → *(1)* *unfamiliar concepts*,
↪ *(2)* *complex theory*,
implementation → *(3)* *'weird" design*,
↪ *(4)* *inadequate ecosystem*, and
applicability → *(5)* *perceived irrelevance*.

This section explains each obstacle category, grouped by the aspect of Agda they are related to. Table I displays the number of entries submitted for each obstacle category as well as the mean and mode of their severity ratings. Figures 3 to 7 illustrate the severity rating distribution of the respective obstacle category, where one dot represents one entry.

Theory-Level Obstacles

Theory-level obstacles refer to the underlying type theory and the programming paradigms on which ITPs are based. They are the obstacles most inherent to dependently-typed programming languages.

When new users first come into contact with a dependently typed language, they have to grasp many *unfamiliar concepts (1)* before they can use it. Learning to program with an inter-

Figure 3: Severity ratings for *unfamiliar concepts (1)*

Figure 4: Severity ratings for *complex theory (2)*

active theorem prover such as Agda presents a big paradigm shift. Not only did participants find dependent types "not intuitive" [P13], but the idea that the "magic happens during type checking instead of execution [takes time] to wrap [their] head around" [P31]. The nature of dependent types forces the programmer to put emphasis on types rather than the actual code. This paradigm shift requires familiarising oneself with new concepts and paradigms, and calls for a different way of thinking than when using more mainstream programming languages.

Additionally, not only are the new concepts unfamiliar, but they are also difficult to grasp due to their underlying *complex theory (2)*. This theory is not hidden behind the design of interactive theorem provers such as Agda, requiring users to understand it in order to be able to reason about the programs they write (e.g., Agda's standard library defines simple pairs in terms of the more complex dependent pairs). "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 or is not able to unify two terms without input from the user). There is simply a lot to learn before one can start writing code in an interactive theorem prover.

Looking at the severity rating distributions in both Figure 3 and Figure 4, it is unclear how serious the theoretical obstacles actually are. Almost every new tool requires learning new concepts and understanding new theory, and we hypothesise that some new users are willing to invest more time in that process. Regardless, more than 20% of all entries pointed to theory-level obstacles, and whether severe or not, they contribute to the steep learning curve of ITPs.

Implementation-Level Obstacles

The second set of obstacles relates to how interactive theorem provers are implemented. This includes everything from the design of their syntax, to their installation process, to the support offered by their main integrated development

```
λx↛x : ∀ {A : Set} → A → A
λx↛x = λ x → x
```

Listing 4: The identity function

environment (IDE). A big advantage of ITPs is the rich type information that is available through interactions with the compiler — something that requires integrated support from the implementation of the language.

The less severe type of obstacle we identified in this domain was Agda's *"weird" [P6] design (3)*. Agda is tailored to theorem proving, which manifests itself in the design of its syntax. Variable names can include almost any Unicode character excluding whitespace[7] [28], allowing mathematicians to write code that resembles handwritten proofs in mathematical notation. As a result, it also allows scenarios like in Listing 4, where the only visual difference between the name and the definition of the identity function are the whitespaces[8].

Unicode characters "[raise the] barrier of entry" [P17] for new users and create confusion during program comprehension. This design choice also imposes a strong requirement of placing whitespaces between all names and operators. Since this is a much stricter requirement than found in most mainstream programming languages[9], new Agda users struggle to adjust to it. Many of our study participants simply wrote "WHITESPACE!" as an obstacle, highlighting their frustrations at this design choice. However, despite being mentioned frequently, the distribution in Figure 5 shows that most participants considered obstacles in this category a minor inconvenience rather than a blocking issue.

The second implementation-level obstacle type that our participants encountered when learning to use Agda was the ITP's *inadequate ecosystem (4)*. With a total of 45 separate entries and a mean severity rating of `5.73`, this category was mentioned most often and rated as most severe. Table II

[7]Whitespace separates names in Agda. Applying the argument `bar` to the function `foo` is expressed as `foo bar`. Agda's parser uses whitespace as an indicator that the name has ended.

[8]Though this is considered poor code style within the Agda community.

[9] `(x,y)`, `(x, y)`, and `(x , y)` would all be parsed as a pair in Haskell.

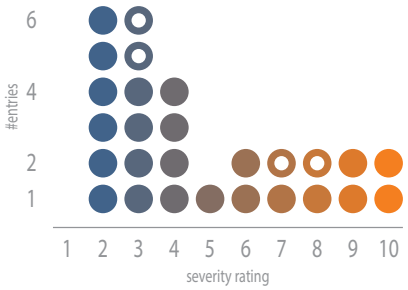

Figure 5: Severity ratings for *"weird" design (3)* (● = problems with syntax; ○ = other)

contains an overview of the severity ratings for interesting instances of this category and Figure 6 shows their distribution.

Our study revealed four issues with the ecosystem that supports Agda:

1) *Tools in Agda's ecosystem are buggy.* While Agda has a wide assortment of tools and features that are intended to make working with it easier, many of them are not of high quality. A common theme (mentioned in 10 separate entries) within the study was that "the installation of Agda is a horrible experience" [P4]. The distribution in Figure 6 show that more than half of the entries about installation were given a high severity rating. Furthermore, the development environment did not always work properly, with complaints that "the Agda plugin in VS Code was often crashing and [the participant] had to restart it" [P18].
2) *The design of Agda's IDE is inconvenient.* The most prominent complaint was about how "syntax highlighting [does not update] automatically and [does not highlight] anything on invalid syntax" [P29], making it "harder to find the mistake" [P15]. While problems with tooling were not rated as a severe instance of this category (with a mean of `4.73`), they were mentioned often, appearing in a total of 15 entries.
3) *The supporting material is incomplete.* There is a "lack of online courses/resources on Agda" [P34] and participants found it difficult to find support when they came across unclear errors or problems. Additionally, some interactive features are under-specified and participants did not fully understand how to use them effectively (e.g., automatic proof search [18]). Although not often mentioned[10], missing and incomplete documentation was rated as the most severe instance of all obstacles categories, with a mean rating of `7.50`. This is not surprising, since Kumar and Chimalakonda identified online tutorials, documentation, and official language resources as the main learning resources for fast-growing programming languages [29, p. 186] — indicating that they are an essential features for languages which aim to grow.
4) *The available support is not accessible to new users.* Error messages are often unclear, require theoretical knowledge and experience to be helpful, and are "almost impossible to understand [in some cases]" [P22]. Of all obstacle instances we found in our responses, error message inadequacy is the most prominent — it contributes 13 entries to its category and a mean severity rating of `7.08`. Additionally, the documentation that is available online is on "super crazy stuff, but very little on the detailed semantics of the basic language" [P5].

Applicability-Level Obstacles

The final category of obstacles we discovered was the participants' *perceived irrelevance (5)* of interactive theorem provers, rated in severity with a mean of `4.83` but with a mode[11] of `7` (see Figure 7). 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 input/output 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.

[10]All information needed to complete the exercises of the course was provided in the custom slides and other course material, meaning it was not usually necessary to search along the public channels.

instance	*#entries*	*mean*	*mode*
frustrating installation	10	5.40	8
buggy & impractical tooling	15	4.73	1
missing documentation	4	7.50	7 / 8
unclear error messages	13	7.08	8

Table II: Severity ratings for interesting obstacle instances of *inadequate ecosystem (4)*

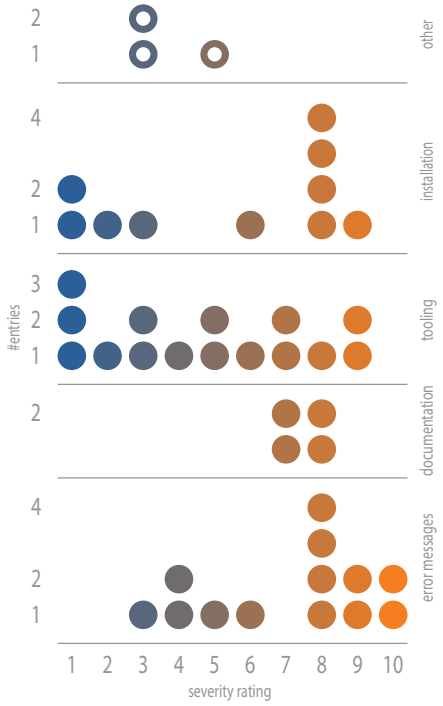

Figure 6: Severity ratings for *inadequate ecosystem (4)*

[11]*Mode* is the most commonly occurring entry in the data.

Figure 7: Severity ratings for *perceived irrelevance (5)*

V. Implications

There are three interesting observations we can make about Agda based on the identified obstacles. These observations correspond to the difficulties listed in Section II.

1) First, *many 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" [30, 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.
2) Second, despite the amount of new, complex theory, *Agda's ecosystem has very little supporting infrastructure for new users*. The syntax is unusual, the use cases differ from more common programming languages, and the documentation is not complete nor 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 new user's difficulty to imagine where Agda could be applied, causes a frustrating onboarding experience.
3) Thirdly, *Agda's design makes it dependent on a custom (and not user-friendly) development environment*. An Agda IDE needs to be able to support features such as interactive commands or writing Unicode characters. Unfortunately, the existing IDEs consist of well-intended features (such as installation through an editor plugin) that do not reach the quality necessary to be practical – they have too many bugs and offer confusing information.

These observations indicate that Agda's design is not being sufficiently considered within the wider context of its own ecosystem. By shifting this focus and taking steps towards making the supporting infrastructure more *robust* and *accessible*, Agda designers could mitigate many of the obstacles that new users face during their learning process.

Robustness would include polishing the existing tools until they are reliable as well as making sure that documentation on important concepts and tutorials for new users are up-to-date and complete. It would resolve frustrating obstacles such as difficult installation and present a smoother learning experience.

Accessibility would entail user-oriented design of the language and its libraries, but also of the learning resources and developer environment. Designing ITPs with usability as a main goal is currently an understudied field, but a crucial one for tools whose underlying theory is inherently abstract and complex.

While these recommendations are based on the obstacles encountered in Agda, designers of all ITPs can use this knowledge as a guide. Examining other ITPs for the same issues and keeping them in mind for the next design iteration can help the community move past these hurdles on to more advanced features and functionalities.

Additionally, we recommend addressing the design of ITPs in the wider context of software engineering. How *can* we use these languages to verify real-world applications? While there has been previous work on using ITPs to verify production code (see Section VII), the integration of these tools into existing software development processes is not obvious. If users truly view ITPs as complementary rather than standalone tools, then it should be clear how they can combine it with others. Clarifying this process entails defining the ITP role and recommended uses within larger projects as well as creating infrastructure which facilitates that integration.

VI. Threats to Validity

We identified three threats to validity in our survey.

First, since the participants were recruited through a course, their responses might have been biased due to expectations of how the survey would affect their grade. To mitigate this effect, we made the survey anonymous and explicitly informed them that participation in this study is entirely voluntary, and that they can withdraw at any time.

Second, before the survey, we had some ideas about which obstacles might be mentioned by the participants. To avoid bias in the responses towards these expected results, we designed the survey to be open-ended and allowed participants to create an entry about any obstacle they wanted.

Lastly, 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 of the course, the timespan they had available for learning, as well as the idiosyncrasies of Agda compared to other ITPs. Though this means that the specific obstacles that were reported might be very particular to this setting, they still provide valuable insight for designers of any ITP as to what could influence the accessibility for new users.

VII. Related Work

Though we have not found any user-studies targeted specifically at dependently-typed interactive theorem provers, there are a number of works that evaluate the usability of ITPs based on other principles:

- Kadoda et al. [30] used a questionnaire based on the "Cognitive Dimensions" framework [31] to evaluate a proof editor meant for educational purposes. The evaluation revealed that proof assistance, meaningful error messages, perceptual cues, and consistency in how classes of rules are handled all have a high effect on the learnability of a proof system.
- Beckert and Grebing [32] also used the Cognitive Dimensions framework to evaluate the KeY System, "a platform of software analysis tools for sequential Java" [33]. They found that important features for usability of the KeY System are proof presentation and guidance, the feedback mechanism, documentation, change management, and the quality of the user interface.
- In later work, Beckert et al. [34] conducted focus groups to detect the usability issues in interactive aspects of

both KeY and Isabelle/HOL [35]. They find that the available automatic proof strategies do not provide users with adequate feedback on how to continue and that technical issues with the interface were annoying to the users.
- Hentschel et al. [36] conducted the first empirical study on ITP user interfaces, comparing the standard UI of the KeY System to a more debugger-like one. They conclude that the debugger-like interface is more effective in presenting the proof steps and that it lowers the barrier to formal verification.
- Grebing and Ulbrich made "design recommendations for user interaction in deductive verification systems" [37]. The recommendations were all on proof writing and comprehension, and gave suggestions about different presentation and interaction methods.

Since the works above focused primarily on the usability of proof writing and comprehension services, they are relevant to ITPs based on any principle. However, unlike our work, they were not executed within the context of a dependently-typed system, and did not take the underlying mathematical theory into account. Additionally, while most of the work above focused on the proof writing mechanisms, we investigated the new user's experience with ITPs and their supporting ecosystems combined. As a result, our findings show the role of the ecosystem and infrastructure as having a stronger influence on the accessibility of ITPs than what is implied in the works above.

Additionally, we use this section to showcase the various attempts at facilitating the use of ITPs within larger projects. These works all take a step towards addressing the design of ITPs in the wider context of software engineering — one of our recommendations from Section V. We can categorise the efforts into three types of integrations:

1) *Translating programs from an ITP.* Some ITPs offer "program extraction" as a way to automatically translate programs from an ITP into more mainstream programming languages (e.g., Coq [38] or Agda [39]). This process allows developer teams to have an expert ITP developer create verified components of a system and then translate them into code that can be integrated into a larger codebase. Coq, for example, offers program extraction into OCaml, Haskell, and Scheme [40]. Similarly, the Agda community has recently brought out a tool called AGDA2HS, which "translates an expressive subset of Agda to readable Haskell, erasing dependent types and proofs in the process" [41]. It requires more annotations within the ITP code than program extraction but promises human-readable output in the target language.
2) *Translating programs into an ITP.* In the opposite direction, the field has seen work on translating code from more mainstream programming languages into a theorem prover, formally verifying the code after it was written. Dybjer et al. [42] and Christiansen et al. [43] have gone through this process manually, translating Haskell code into Agda and Coq respectively. An automatic translation method from Haskell to Agda has been proposed by Abel et al. [44], and tools such as `hs-to-coq` [45] or `coq-of-ocaml` [46] already exist for Coq.
3) *Adding refinement types to existing programming languages.* Though less powerful than a dependently-typed system, refinement types can "extend [a programming language] with predicates that restrict the allowed values in variables and methods" [47]. The most mature implementation of refinement types is in Liquid Haskell [48], which uses an SMT solver and Liquid Proof Macros ("an extensible metaprogramming technique" [49]) to write proofs about Haskell code. We have seen similar attempts in Java [47], JavaScript [50] / TypeScript [51], and Scala [52] — though they are still in early stages of implementation.

All of these approaches have drawbacks:

1) translating from an ITP requires implementations of target language libraries in the source theorem prover,
2) translating to an ITP allows only post-hoc verification[12], becoming essentially error-detection tools (albeit more rigorous ones than usual), and
3) refinement types are less powerful than dependent types.

Despite that, all three approaches provide a good starting point for future work on integrating ITPs into existing software development workflows.

VIII. Conclusion & Future Work

Interactive theorem provers promise strong guarantees about program correctness and could be a powerful tool for improving software quality. In our survey, conducted with 41 bachelor students who were learning to use Agda, we learned that new users can see the potential benefits of using ITPs, but they struggle to understand their role and relevance within software development. Our results indicate that they view ITPs as supplementary rather than a standalone tools. Unfortunately, we saw that the ecosystem that supports Agda and could potentially help integrate it into software development processes is *buggy*, *inconvenient*, *incomplete*, and *not accessible to new users*. Additionally, new users struggle with Agda's mathematics-oriented design, the *unfamiliar concepts* and *complex theory* that ITPs are based on, and the uncertainty of their role and relevance within software development.

Based on these results, we recommend viewing the design of Agda within the wider context of its own ecosystem and that of existing processes in software engineering instead of focusing primarily on the specifications and design of the language itself. Putting emphasis on the robustness and accessibility of Agda's infrastructure can mitigate many of the obstacles that new users face when learning to use the language, and would open up avenues for research in the uncharted field of user-oriented ITP design. Additionally, considering the role and recommended uses of ITPs within larger, multilingual projects

[12]Post-hoc verification is when properties are only verified after the program has been written, as opposed to integrating the specifications into the types.

and facilitating that integration with suitable infrastructure would likely make it easier to adopt them as verification tools in real-world settings.

Although these recommendations are based on obstacles found in Agda, we believe they can serve as guides for designers of any interactive theorem prover, helping them avoid those obstacles and steer towards a better learning experience. Conducting similar studies on a wider selection of ITPs and participants might provide richer and more varied insights. Furthermore, resolving this initial set of obstacles might reveal the next set of ITP design challenges which can be studied in future work (e.g., we anticipate that there are interesting usability-related questions in the design of the ITP interactivity). By continuing this line of work, we can pave the way for more easily achievable formally verified software.

Acknowledgments

This research is partially sponsored by the Dutch Science Foundation NWO through the Vici "TestShift" project (No. VI.C.182.032). Jesper Cockx holds an NWO Veni grant on 'A trustworthy and extensible core language for Agda' (VI.Veni.202.216).

References

[1] A. J. Ko, B. Dosono, and N. Duriseti, "Thirty years of software problems in the news," in *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2014, pp. 32–39. [Online]. Available: https://doi.org/10.1145/2593702.2593719

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606617

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 202–211. [Online]. Available: https://doi.org/10.1145/2597073.2597082

[4] M. Aniche, *Effective Software Testing: A developer's guide*. Manning, 2022.

[5] M. Aniche, C. Treude, and A. Zaidman, "How Developers Engineer Test Cases: An Observational Study," *IEEE Trans. Software Eng.*, pp. 4925–4946, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3129889

[6] B. C. Pierce, *Types and Programming Languages*. Cambridge, Massachusetts: The MIT Press, 2002.

[7] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, 1969, publisher: American Mathematical Society. [Online]. Available: https://doi.org/10.2307/1995158

[8] S. Klabnik and C. Nichols, "Understanding Ownership," in *The Rust Programming Language*, 2nd ed. San Francisco, CA, USA: No Starch Press, 2021.

[9] U. Norell, "Towards a practical programming language based on dependent type theory," PhD Thesis, Chalmers University of Technology, Göteborg, Sweden, 2007.

[10] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992, conference Name: Computer. [Online]. Available: https://doi.org/10.1109/2.161279

[11] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A Human Study of Comprehension and Code Summarization," in *International Conference on Program Comprehension (ICPC)*. ACM, 2020, pp. 2–13. [Online]. Available: https://doi.org/10.1145/3387904.3389258

[12] J. Harrison, J. Urban, and F. Wiedijk, "History of Interactive Theorem Proving," in *Handbook of the History of Logic*, ser. Computational Logic. North-Holland, 2014, vol. 9.

[13] The Coq Development Team, "The Coq Proof Assistant," Sep. 2024. [Online]. Available: https://zenodo.org/records/14542673

[14] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean Theorem Prover (System Description)," in *Automated Deduction (CADE-25)*, A. P. Felty and A. Middeldorp, Eds. Springer International Publishing, 2015, pp. 378–388. [Online]. Available: https://doi.org/10.1007/978-3-319-21401-6_26

[15] A. Stefik and S. Hanenberg, "Methodological Irregularities in Programming-Language Research," *Computer*, vol. 50, no. 8, pp. 60–63, 2017. [Online]. Available: https://doi.org/10.1109/MC.2017.3001257

[16] E. Brady, *Type-Driven Development with Idris*. Manning Publications Co., 2017.

[17] The Agda Development Team, "Emacs Mode: Commands in context of a goal." [Online]. Available: https://agda.readthedocs.io/en/v2.7.0.1/tools/emacs-mode.html#commands-in-context-of-a-goal

[18] ——, "Automatic Proof Search (Auto)." [Online]. Available: https://agda.readthedocs.io/en/v2.7.0.1/tools/auto.html

[19] S. Fu, T. Dwyer, P. J. Stuckey, J. Wain, and J. Linossier, "ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 146–156. [Online]. Available: https://doi.org/10.1109/ICPC58990.2023.00029

[20] E. Babbie, *The Practice of Social Research*, thirteenth edition ed. Wadsworth, Cengage Learning, 2013.

[21] H. Taherdoost, "What Is the Best Response Scale for Survey and Questionnaire Design; Review of Different Lengths of Rating Scale / Attitude Scale / Likert Scale," *International Journal of Academic Research in Management*, vol. 8, no. 1, pp. 1–10, 2019.

[22] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9150-8

[23] E. Babbie, *The Basics of Social Research*, fourth edition ed. Thomson Higher Education, 2008.

[24] R. Hoda, *Qualitative Research with Socio-Technical Grounded Theory: A Practical Guide to Qualitative Data Analysis and Theory Development in the Digital World*. Springer Cham, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-60533-8

[25] M. B. Miles and M. Huberman, *Qualitative data analysis: an expanded sourcebook*, 2nd ed. Thousand Oaks, CA: Sage, 1994.

[26] K. Charmaz, *Constructing Grounded Theory*, 2nd ed., ser. Introducing Qualitative Methods. Sage Publications, 2014.

[27] B. G. Glaser, *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Press, 1978.

[28] The Agda Development Team, "Lexical Structure: Keywords and special symbols." [Online]. Available: https://agda.readthedocs.io/en/v2.7.0.1/language/lexical-structure.html#keywords-and-special-symbols

[29] J. Kumar and S. Chimalakonda, "What Do Developers Feel About Fast-Growing Programming Languages? An Exploratory Study," in *International Conference on Program Comprehension (ICPC)*. ACM, 2024, pp. 178–189. [Online]. Available: https://doi.org/10.1145/3643916.3644422

[30] G. F. Kadoda, R. G. Stone, and D. Diaper, "Desirable features of educational theorem provers - a cognitive dimensions viewpoint," in *Annual Workshop of the Psychology of Programming Interest Group*, 1999.

[31] 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, 1996. [Online]. Available: https://doi.org/10.1006/jvlc.1996.0009

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

[33] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, "The KeY platform for verification and analysis of Java programs," in *Verified Software: Theories, Tools and Experiments*. Springer, 2014, pp. 55–71. [Online]. Available: https://doi.org/10.1007/978-3-319-12154-3_4

[34] B. Beckert, S. Grebing, and F. Böhl, "A Usability Evaluation of Interactive Theorem Provers Using Focus Groups," in *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2015, pp. 3–19. [Online]. Available: https://doi.org/10.1007/978-3-319-15201-1_1

[35] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[36] M. Hentschel, R. Hähnle, and R. Bubel, "An empirical evaluation of two user interfaces of an interactive program verifier," in *International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 403–413. [Online]. Available: https://doi.org/10.1145/2970276.2970303

[37] S. Grebing and M. Ulbrich, "Usability Recommendations for User Guidance in Deductive Program Verification," in *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich, Eds. Springer International Publishing, 2020, pp. 261–284. [Online]. Available: https://doi.org/10.1007/978-3-030-64354-6_11

[38] P. Letouzey, "A New Extraction for Coq," in *Types for Proofs and Programs*. Berlin, Heidelberg: Springer, 2003, pp. 200–219. [Online]. Available: https://doi.org/10.1007/3-540-39185-1_12

[39] The Agda Development Team, "Compilers." [Online]. Available: https://agda.readthedocs.io/en/v2.7.0.1/tools/compilers.html

[40] The Coq Team, "Extraction of programs in OCaml and Haskell." [Online]. Available: https://coq.inria.fr/doc/V8.11.1/refman/addendum/extraction.html

[41] J. Cockx, O. Melkonian, L. Escot, J. Chapman, and U. Norell, "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs," in *International Haskell Symposium*. ACM, 2022, pp. 108–122. [Online]. Available: https://doi.org/10.1145/3546189.3549920

[42] P. Dybjer, Q. Haiyan, and M. Takeyama, "Verifying Haskell programs by combining testing, model checking and interactive theorem proving," *Information and Software Technology*, vol. 46, no. 15, pp. 1011–1025, 2004. [Online]. Available: https://doi.org/10.1016/j.infsof.2004.07.002

[43] J. Christiansen, S. Dylus, and N. Bunkenburg, "Verifying effectful Haskell programs in Coq," in *International Symposium on Haskell*. ACM, 2019, pp. 125–138. [Online]. Available: https://doi.org/10.1145/3331545.3342592

[44] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell, "Verifying haskell programs using constructive type theory," in *Workshop on Haskell*. ACM, 2005, pp. 62–73. [Online]. Available: https://doi.org/10.1145/1088348.1088355

[45] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total Haskell is reasonable Coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. ACM, 2018, pp. 14–27. [Online]. Available: https://doi.org/10.1145/3167092

[46] Guillaume Claret, "coq-of-ocaml." [Online]. Available: https://ocaml.org/p/coq-of-ocaml/2.5.3%2B4.14

[47] C. Gamboa, P. Canelas, C. Timperley, and A. Fonseca, "Usability-Oriented Design of Liquid Types for Java," in *International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1520–1532. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00132

[48] N. Vazou, J. Breitner, R. Kunkel, D. Van Horn, and G. Hutton, "Theorem proving for all: equational reasoning in liquid Haskell (functional pearl)," in *International Symposium on Haskell*. ACM, 2018, pp. 132–144. [Online]. Available: https://doi.org/10.1145/3242744.3242756

[49] H. Blanchette, N. Vazou, and L. Lampropoulos, "Liquid proof macros," in *International Haskell Symposium*, ser. Haskell 2022. ACM, 2022, pp. 27–38. [Online]. Available: https://doi.org/10.1145/3546189.3549921

[50] R. Chugh, D. Herman, and R. Jhala, "Dependent types for JavaScript," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*. ACM, 2012, pp. 587–606. [Online]. Available: https://doi.org/10.1145/2384616.2384659

[51] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for TypeScript," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016, pp. 310–325. [Online]. Available: https://doi.org/10.1145/2908080.2908110

[52] G. S. Schmid and V. Kuncak, "SMT-based checking of predicate-qualified types for Scala," in *Proceedings of the ACM SIGPLAN Symposium on Scala (SCALA)*. ACM, 2016, pp. 31–40. [Online]. Available: https://doi.org/10.1145/2998392.2998398