Programs and Proofs in Practice

A Grounded Theory on How People Program with Dependent Types

Ruben Backx
Delft University of Technology
The Netherlands
R.W.Backx@tudelft.nl

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

Wouter Swierstra
Utrecht University
The Netherlands
W.S.Swierstra@uu.nl

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

Abstract

Dependently typed programming languages (DTPLs) are powerful tools that help improve software correctness. However, they currently serve niche communities, and have not yet been adopted in mainstream software development processes. Through this work, we aim to understand *how people program in dependently typed languages* to provide a solid, layered understanding of the starting point for any future advancements in DTPL design. We apply the *small-scale*, *emergent-mode* Socio-Technical Grounded Theory methodology to develop a theory grounded in data from interviews conducted with DTPL users with varying levels of experience, using Agda, Rocq, and Lean. Our results show how DTPL programmers use types to plan and constrain their programs; make trade-offs in how they represent data; and continuously interact with the compiler to construct a program iteratively. Based on our findings, we have formulated five recommendations for the developers and maintainers of DTPLs who wish to make their work more accessible to a broader audience of software developers.

1 Introduction

Dependently typed programming languages (DTPLs) such as Agda [33], Rocq [40], and Lean [12] provide a unified system for programming and formal verification, and have gained traction within the academic community. It is not uncommon for papers on programming languages to be accompanied by some form of mechanised proof; benchmarks such as the POPLMark challenge [3] have helped push the further development of these systems. Lean has gathered momentum among mathematicians. Mathlib, a community-built library of formalised mathematics, now contains mechanised proofs of tens of thousands of theorems, formalising both a substantial part of undergraduate mathematics and recent research results. The CompCert project uses Rocq to verify a real-world C compiler [26]. Yet despite all these successful academic applications of DTPLs, their impact on day-to-day software engineering remains much more limited.

Since the rich contextual information that dependent types provide lends itself well to interactive theorem proving, the most popular DTPLs have been designed to support program writing and interactive verification in one unified environment. In principle, these systems are therefore prime candidates for implementing safety-critical software, where failure has severe consequences. In practice, however, DTPLs have not (yet) been widely adopted by industry. Instead, most software is verified using methods that are more integrated into mainstream development processes, such as code review or automatic software testing. The shared understanding of the community is that the DTPL learning curve is very steep, and that the cost of using DTPLs for verification is too high for day-to-day industry use. Previous research has shown that new users find it hard to imagine writing software that end users can interact with in a DTPL [23, p. 166].

How can we make DTPL technology more accessible to a wider range of programmers? In order to provide recommendations to DTPL maintainers with such goals, we first need a good understanding of how people *currently* use DTPLs. Research in this direction has picked up over the last few years, with Shi et al. [36] conducting an observation study on how people write proofs in DTPLs with tactics, and Juhošová et al. [23] identifying the obstacles that new users face when learning to use Agda. In this work, we hope to generalise their findings by viewing DTPLs as programming languages rather than just theorem provers and by focusing on users with varying levels of experience with DTPLs. We answer the following research question:

> **RQ:** How do people program in dependently typed languages?

We answer this question by applying Socio-Technical Grounded Theory (STGT) [20, 21] to twelve interviews conducted with DTPL users from different backgrounds, working in Agda, Rocq, or Lean. Grounded Theory as a methodology is becoming more popular in software engineering [2, 22, 29, 34, 37, 44] and programming languages [31] research

to study the human aspects of these fields. STGT is a particularly good methodology for answering "how" questions and is best suited to "studying topics that are socio-technical in nature and where the research team wishes to produce rich and layered qualitative findings" [21, p. 45]. Such rich, layered findings allow us to provide a full theory on how people currently use DTPLs, which in turn allows us to provide data-grounded recommendations for making DTPLs more accessible to a wider range of developers.

Our results present a theory consisting of seven hypotheses, explaining how DTPL programmers use types to plan and constrain their programs, make trade-offs in design decisions, and interact with the compiler[1]. We use these hypotheses to provide five recommendations for DTPL maintainers and developers to help them make their work accessible to a broader audience, and discuss the work that has already made progress in these directions. By grounding our recommendations in the collected data, we provide a user-oriented basis for future work in DTPL usability. We view this work as a stepping stone towards bringing dependently typed programming into mainstream development processes.

2 Methodology

To answer our research question, we used Socio-Technical Grounded Theory [21], a methodology for forming new theories grounded in (usually qualitative) data. In this *small-scale*, *emergent-mode* version of such a study, we conducted *online* think-aloud programming sessions and semi-structured interviews with twelve DTPL users. We analysed the data using our *constructivist* research paradigm as a team of researchers focusing on dependent type theory, functional programming, and the usability of advanced type systems. The resulting theory is presented in Section 3 as a set of hypotheses supported by quotes from the interviews, and we provide recommendations for DTPL maintainers based on our findings in Section 4.

2.1 Socio-Technical Grounded Theory

Socio-Technical Grounded Theory (STGT) is a version of the Grounded Theory methodology [6, 11, 16] defined by Hoda [20, 21] for projects that study *socio-technical* phenomena. It consists of a basic and advanced stage, producing preliminary hypotheses and a mature theory respectively. Grounded Theory is applied in iterations, where each iteration consists of the research team (1) collecting new data, (2) analysing the new data using *coding* and *constant comparison* to organise it into concepts and categories, and (3) deciding where to collect data next. The process stops upon reaching *theoretical saturation* [16, p. 61], i.e, when an iteration does not provide any new insights. Throughout the entire process, the team writes memos — notes of "ideas about codes and their relationships as they strike the analyst" [15] — and use them to discuss and form the theory.

Coding is "the process of closely inspecting, deeply making sense of, and inferring meaning from data and giving those meanings some labels" [21, p. 232]. During the basic stage, we used *open coding*, labelling the data "inductively and comprehensively, with an open mindset, to enable emergence of information and insights, without looking to find anything specific in the data" [21, p. 233]. For example, the following interview snippet was coded as "`difficulty refactoring`":

> [P3] Agda is very good at this exploratory stuff, but when it comes to maintenance and refactoring…

Once we started seeing emergent categories and hypotheses, we moved on to *targeted data collection* [21, p. 299], using *targeted coding* to focus on only the most significant / promising concepts and categories. The process was still inductive, but fresh codes were constantly compared with other codes, and similar codes were grouped into concepts. For example, during a subsequent interview, we added the following quote into the "`refactoring`" concept:

> [P7] I just tried to make things as simple as possible and build up on top because it's really inflexible to work with it.

This concept was later grouped into the "`ecosystem`" category, along with concepts such as "`standard library`", "`documentation`", and "`error messages`". These concepts, codes, and quotes formed the basis for our hypotheses and recommendations, presented in Sections 3 and 4.

Because it was manageable for a single analyst to code the dataset, the whole coding process was conducted by the first author to ensure consistency. However, as suggested by Hoda [21, p. 260], the quotes and codes were regularly reviewed by the other researchers on the team, and the resulting concepts and categories were discussed in weekly meetings. We acknowledge the creative subjectivity of our codebook (provided in the data repository [4]), declare our *constructivist* research paradigm [21, p. 46], and discuss how this approach might have affected our results in Section 5.

2.2 The Interviews

We collected our data in four iterations of *online* interviews with a total of twelve participants. An overview of their background, the editor they used, and how they were recruited is provided in Table 1. The interviews consisted of an hour-long think-aloud session during which they used the DTPL and a subsequent 30-minute *semi-structured* interview [21, pp. 177–188]. They were conducted over Zoom, and the recordings stored on GDPR-compliant servers. Participants were asked to sign an informed consent form, agreeing to have their voice and screen recorded, and the anonymised transcripts of their interviews published [4].

The interview recordings were transcribed using Whisper [35], diarised using Whisper X (run locally), and then checked and corrected manually. We did not transcribe the

[1] By "compiler" we mean the tool that parses, scope checks, type checks, and compiles or interprets the code.

id	identified as	invite	editor
P1	Formal Methods Engineer	personal	Emacs
P2	Researcher (type theory)	personal	Emacs
P3	Researcher (type theory)	personal	Emacs
P4	Student (BSc Computer Science)	personal	VS Code
P5	Student (BSc Computer Science)	personal	VS Code
P6	Student (BSc Computer Science)	personal	VS Code
P7	Student (BSc Computer Science)	personal	VS Code
P8	Software Developer	personal	VS Code
P9	Researcher (mathematics)	online	VS Code
P10	Researcher (mathematics)	online	VS Code
P11	Researcher (type theory)	personal	VIM
P12	Researcher (type theory)	online	VS Code

Table 1. An overview of the participants

think-aloud sessions, but instead wrote down whenever the participant did anything interesting, and then asked about it during the interview. This meant that the think-aloud data was analysed as part of the interview, and the original recording was referenced if additional details were needed. After a transcript was fully processed, we rewatched the full recording and wrote down and coded anything we missed.

Pilot. The first two interviews were conducted as pilots with participants who were experienced Agda users. We asked them to have their own installation of Agda ready and to work in the environment they always do when using the DTPL. We gave them 45 minutes to implement an assignment where, given a valid, partially-solved Sudoku and the number a players fills in, they had to determine whether the resulting Sudoku is still valid. The participants were encouraged to approach the assignment however they saw fit. We chose a programming problem instead of proof-writing one such that we could study Agda when used as a programming language rather than a theorem prover. The participants were asked to share their screen and to voice all their thoughts while programming. We prepared a rough outline for the subsequent 45-minute interview, focusing on the programming setup they use, the different features and use cases of Agda, and the challenges in the assignment.

We found that the recording setup worked well, and that we were gathering the type of information we were looking for. However, we did notice that the participants found the allocated 45 minutes too short for the think-aloud session, and adjusted the schedule to 60-minute think-aloud sessions and 30-minute interviews. The outline of the interview stayed roughly the same, but we decided to drop discussions on their programming setup, since we could derive those ourselves from the recordings. We did not ask each participant every question and preferred specific questions relating to the think-aloud part of the interview. The interview protocol and question template is available in our data repository [4].

Round 1: Experienced Agda Users. Since the pilot interviews went well, we used them as data in the first round, and conducted one more interview to round up the data collection of the first STGT iteration. This meant that the first set of participants were experienced Agda users, all recruited via personal invite. After using the basic stage of STGT to analyse the data collected from this first round, we already saw a variety of hypotheses emerging with two avenues of interest for further data collection: (1) the participants struggled with completing the Sudoku assignment we had given them even though we thought it would be relatively simple; and (2) the participants seemed to have an aversion to using the standard library in their solutions. Since the concepts and categories were already forming relatively strong emerging hypotheses, we moved on to the advanced stage of STGT and targetted participants who could fill in the gaps.

Round 2: New Agda Users. The participants from the first round had indicated that they spent a large portion of their time planning their program, leaving them with less time to actually focus on solving the assignment. We suspected that this was because they were already thinking about the design decisions that would make verification easier in the long run, and decided to explore that hypothesis by interviewing users who were new to DTPLs. We invited a set of four computer science students who had just had a two-week introduction to Agda at our university. Notably, these new users were actually able to get *further* in the Sudoku assignment than the professional participants in terms of functionality, but they did so by using almost none of Agda's dependently typed features, creating a program with few, if any, proofs. We also noticed that when the students *did* try to use a dependently typed feature, they struggled more than the experienced users and required more planning.

Round 3: Mathematicians Using Lean. To explore the aversion to using the standard library in Agda, we decided to invite three users of Lean, a more recently developed DTPL. We had heard that Lean has a good ecosystem, active community engagement, and a popular library for mathematics called Mathlib. Unlike Agda, proving in Lean is usually done using tactics, which meant that our concepts and categories expanded to account for this different proving approach. Additionally, many of Lean's users are mathematicians looking to formalise their theorems, and we decided to include them in this round of interviews, investigating how users who are not necessarily computer scientists interact with a DTPL. At this point, we also evaluated that there was no more to learn from the Sudoku assignment, and we asked each participant to work on their own projects during their think-aloud sessions, allowing us to observe how a DTPL is currently applied in real-world scenarios. We reached out to interested

participants with the relevant experience who had filled in an online sign-up survey distributed over our social media and via DTPL community channels.

Round 4: Rocq Users. For the fourth round of interviews, we invited two Rocq users. This was done to verify that our hypotheses are robust and extend to a wider range of DTPLs. One participant was invited via personal invite and one was invited via the online sign-up. In this final iteration, we made minimal adjustments to our categories, and were able to form our mature theory.

3 Results

Based on our data, we were able to form seven overarching hypotheses about how people program in dependently typed languages. These are related to how DTPL programmers plan and design their programs, how they interact with the DTPL, and the control they wish to have over certain features. The anonymised transcripts of the interviews are available under an MIT Licence [4].

3.1 Planning

H1: DTPL users use types to create a plan for their program and to constrain the solution space, adjusting it as necessery throughout the process.

During the first round of interviews, we gave participants what we assumed to be a relatively simple programming assignment, asking them to validate an insertion into an already-valid Sudoku. Surprisingly, we found that most expert participants started over-engineering their solutions, creating highly complex data types considering the task at hand, and did not get as far as we expected. This lead us to give the same assignment to the participants from the second round, who were new to using a DTPL, but not new to programming. As expected, they planned less and implemented more functionality within the same time frame, but remarked that they could have made more progress using a programming language with which they were more familiar. The expert-level participants explained that they had learned to carefully plan their programs before they start writing any code, in order to choose a definition that will not require major refactoring later. The students also remarked that they needed to think further ahead than in other programming languages.

> [P7] If this were any other language, I would just type some stuff. It wouldn't work but, immediately, I would be able to fix it. But here, I need to think ten steps in advance.

Participants planned their solutions by sketching outlines for their programs with data types and type signatures, and then revised and refined them throughout the entire think-aloud session. They claimed that thinking far ahead is necessary when programming in a DTPL, as the choice of the "plan" — and therefore the choice of the (data) types — can determine the rest of the program. The students we interviewed considered this tight coupling between types and implementation to be a downside, but expert participants overwhelmingly saw this as a benefit.

The type information can be leveraged by adopting some style of "hole-driven development", which many participants did. In this style of development, the programmer scans through the program and looks for a "hole"[2] they want to fill based on the local context and goal. They then attempt to fill the hole or, if that proves too difficult, fill the hole *partially*, i.e., by filling it with an expression containing more holes. Alternatively, the programmer can revise the types to make it easier to fill the hole. Revision can be the only viable option if the programmer's original plan turns out to be flawed — which happened in almost every think-aloud session.

> [P2] Holes are super useful. Also, when I'm programming Haskell, I program in the same style now. [...] Because, typically, the types tell you where to go.

Participants told us that this process of filling holes and revising the types feels like solving a puzzle, which makes using a DTPL not only useful, but also fun. They found formalising a proof similar to playing a game, where strategy plays a big role in determining how smooth the process is going to be. If they eventually find out that their strategy was not designed well, they have to start refactoring their code — a point in which the process can stop being fun.

> [P3] Okay, I want to change something in my data type. Maybe I want to, you know, add a field to a constructor or remove a field from a constructor or add a constructor to a data type, and then you end up having to do all this manual work, which is too bad.

Refactoring, they explained, is typically not an enjoyable task in a DTPL. If a type signature needs to be adjusted (e.g., the type of an argument needs changing or a constructor has to be added to a data type), the programmer has to go through the entire code base and manually fix all the places where that signature is used. According to our participants, the current tooling does not help with automating such a scenario. They consider this to be an oversight.

3.2 Design

H2: DTPL users aim for a "sweet spot" between intrinsic and extrinsic verification, influenced by the design of the DTPL they are using.

With "intrinsic" and "extrinsic" we mean the degree to which required invariants are embedded in the data types. An approach where properties are represented and verified extrinsically offers simple data types, but decoupling the properties from the definitions has its drawbacks. For example, establishing certain invariants may rely on additional

[2] A hole acts as placeholders for not-yet-implemented parts of the code and is denoted using a question mark (?) in Agda and `sorry` in Lean.

properties that need to be proven and invoked separately. As a result, programmers need to know where to find the auxiliary properties their proof requires, which can be challenging if there are many places to look. More frustratingly, functions may still require the programmer to implement "bogus" cases for the sake of totality, even though the extrinsic property clearly shows those cases to be impossible.

When taking a more intrinsic approach, where the proofs are stored directly in the data types, the programmer can easily find the proofs they need and does not need to provide implementations for impossible program branches. However, by requiring all instances of a data type to contain a proof, the possible variants that can be created are constrained. This can be beneficial, but can also get the programmer stuck — especially when re-establishing the intrinsic proofs is non-trivial. If it turns out that the invariant is too strict for a certain scenario or needs to be temporarily broken, the code needs to be refactored to accommodate for this. This point is illustrated by what some participants did with the Sudoku assignment. Some experienced participants first designed their solution in an intrinsic manner, for example:

```
record Sudoku : Set where
  field
    grid : Vec (Vec (Maybe (Fin 9)) 9) 9
    noDuplicatesInRows : (i : Fin 9) → lookup
      (noDuplicates (rows grid)) i ≡ true
    noDuplicatesInColumns : ...
    noDuplicatesInBoxes : ...
```

This definition encodes the "no duplication" properties directly into the structure holding the data that represents the current state of the Sudoku. The participants would then attempt to solve the assignment by first applying a move to the Sudoku, and only then checking whether the Sudoku is still valid. They quickly came across the problem: applying a potentially invalid move to a Sudoku is not possible using a representation that allows only valid Sudokus to be constructed. After realising this, the participants moved towards a more extrinsic approach:

```
record Sudoku : Set where
  field
    grid : Vec (Vec (Maybe (Fin 9)) 9) 9

valid : Sudoku → Bool
valid sudoku = all noDuplicates (rows sudoku)
            && ...
```

Most participants ended up with this representation.

[P1] Probably you could encode valid Sudokus in Agda somehow. It might also be a very ugly and unwieldy representation. That's one end of the spectrum and the other end of the spectrum is just having a bunch of lists. And I think there's something in between there.

Where exactly the ideal solution lies on the spectrum depends on the DTPL. Generally, a DTPL that enables easily searching for proofs, such as Lean, is going to lean more towards the extrinsic side of the spectrum, whereas a DTPL that is designed more around embedding the right properties in the data, like Agda, leans more towards the intrinsic side. Some participants mentioned the existence of a certain "sweet spot" on the spectrum of representing invariants, specific to the problem at hand and the DTPL used to solve it. They were looking for such a sweet spot multiple times during the think-aloud sessions.

H3: DTPL users search for the "right level" of abstraction in their code.

Across all the DTPLs, programmers indicated that there is a certain level of abstraction they aim for when designing their data types and proofs, differing per task and DTPL. Participants explained that, when contributing to a library, whether intended for personal use or to be shared with the community, they often aim for a higher level of abstraction to ensure the proofs can be applied in a wider range of contexts. While many participants did not start out with such a goal, they often realised the proof or function they were working on might be useful in the future, leading them to refactor their work so that it would be more abstract. For example, one participant started out with the following definition for a function that splits a vector into equal parts:

```
chunk : {A : Set} → Vec A 9 → Vec (Vec A 3) 3
```

They later refactored it into a more generally applicable definition:

```
chunk : {A : Set} → {n m : Nat}
      → Vec A (n * m) → Vec (Vec A m) n
```

Most participants started refactoring towards more abstract representations only once they had a complete implementation of their main goal. When *defining* the main goal or a sub-goal, however, a lot of thought was put into determining the right level of abstraction.

[I] "Do you think that the generalisation actually gives you better insight into the problem?"

[P1] "Yes [...] it's also going to be a shorter solution if you do it more generally."

[I] "Do you ever find yourself reusing your old code that you've made more general?"

[P1] "Yes."

It should be noted that the Sudoku assignment description did not require the solution to support any size other than 9×9. Even if a participant chose to hardcode some part of their solution, they often made some other part more abstract than it needed to be, e.g., by having a `rows` function work for any size of list. In later rounds of interviews, where

participants worked on their own projects, they also made some parts of their work more abstract than it needed to be.

> [P3] And then I did this really icky thing, which is I knew it's like three, so I could just pattern match on it and that was very unsatisfactory for me because it's like, oh, I'm hacking my way through here.

Despite this preference towards abstraction, most participants also seemed to have a notion of a solution that would be "too abstract". Participants considered a lemma or function too abstract when it was no longer recognisable as a solution to the specific problem it was intended to solve.

Both the field in which a participant worked and the DTPL they used contributed to their understanding of the "right" level of abstraction. In mathematical disciplines, participants followed the conventions of the Mathlib lemmata defined for that discipline. More programming-oriented tasks that did not require as many proofs were allowed to be more concrete. The most abstract definitions were found in type-theoretical work, where participants often leveraged the type system of DTPLs to their full extent. Most Agda programmers told us they have a personal library of definitions. Any definition they considered to be a nice addition to their library would be written using a higher level of abstraction.

H4: DTPL users follow a code style geared towards maintainability and community conventions.

Besides ending up with a compiling program, our participants cared about a secondary goal: their code looking "nice".

> [P10] So, at the end, of course, you want a proof that works, because you want a proof, and there's no proof that doesn't work, by definition of proof. But you want a nice proof, you want an elegant proof. I mean, just having a proof is the bare minimum.

What exactly "nice" meant to each participant differed slightly and was influenced by the DTPL they were using. Most participants agreed that a short proof was a good proof. However, participants using the more tactic-oriented DTPLs were also wary of making a proof *too* short, fearing that they would make it unintelligible by using too complicated functions and tactics. For Agda users, the most prominent opinion seemed to be "the shorter, the better".

Our participants explained their preferences were mostly guided by legibility and maintainability. In Agda, a shorter proof is often more elegant since language elements map roughly one-to-one to reasoning steps. However, since tactics can perform arbitrarily complex transformations of the proof state, a short proof in a DTPL with tactics offloads much of the reasoning to the tactic implementation. If the tactics involved are complex, e.g., performing automated proof search, it can be hard to comprehend the proof structure.

> [P10] What I would call a nice proof, at least a nice Lean proof, is a proof that is at the same time very short, but contains all the relevant information to understand how the argument went. So if it becomes too short, so it hides some ideas or some insight [...], then it's not a nice proof.

Another style consideration for most participants was whether to use external libraries. Most participants did not want to use unofficial libraries, even if writing their own implementations required more work, usually because the libraries were difficult to install or because they did not use the same code style as the participant. There was, however, a division between the Agda users and the rest on whether to use the standard library[3]. Lean and Rocq users expect others to be familiar with the standard library, so using it makes their own code more understandable. Agda's standard library, on the other hand, is not as widely used, and not all Agda programmers are familiar with its content or code style. As a result, Agda users prefer putting a custom definition in the same file, keeping the development more self-contained and easy to understand.

Finally, the tactic-oriented DTPLs had an additional code style consideration which dictates that "powerful" tactics — that is, tactics with a high degree of automation — should not be used in the middle of a proof. Such tactics might be able to do more than solve the current sub-goal in the proof; if any later refactoring changes the sub-goal at that point, the tactic might change the proof state in unexpected ways, potentially causing a confusing error later down the proof. Participants who *did* use a powerful tactic in their first implementation (such as `simp` in Lean) later replaced it with a restricted version that would do only exactly what was expected (such as `simp only` or `apply`). Powerful tactics at the end of a proof were not considered a problem, as they cannot affect any later lines of the proof.

3.3 Interactivity

Several participants used the term "conversation" to refer to their interactions with the DTPL.

> [P2] I feel like working with Agda is like having a conversation with a partner, right?

H5: DTPL users write code by conversing with the compiler, which can act as an oracle, but also request more information from the user.

With the combination of restrictive types and interactive holes, programming in a DTPL can often feel like conducting a conversation with the compiler. This interactivity begins with a (partial) type definition and a hole which can help guide the implementation process and break it down into

[3] Mathlib is so commonly used in Lean that we consider it to be a "standard" library for the purposes of this paper.

smaller steps. A series of such steps then guides the programmer to a complete solution, or signals when further progress is impossible and something needs to be adjusted.

Using holes, the programmer can ask the compiler for local type information ("What is the type of this variable?"), request real-time feedback on potential solutions ("Can I apply this tactic here?"), or even delegate the solution search to the compiler ("Can you split this variable into cases?"). Our participants knew that the system knows more about the exact types in their program than they do. They frequently used this to their advantage by, for example, using a keyboard shortcut to explicitly ask the system for a target type or by referring to the window displaying components currently available in the context.

[P1] I rely a lot on the type checker.

Conversely, the compiler sometimes requests more input from the programmer, usually by providing an error message. For example, if the programmer does not apply a function to the correct number of arguments, the Agda compiler would report "`Error: [UnequalTerms] X → Y !=< Y when checking that the inferred type of an application matches the expected type`", as a prompt for the programmer to supply more arguments. In more obscure error messages, the system can simply report "`Error: unsolved meta variable`" followed by a list of generated unification constraints, essentially telling the user it cannot automatically infer the correct type without more information. Similarly, the following Lean snippet produces "`don't know how to synthesize implicit argument 'n' @Vector.mk Nat ?m.29 { toList := [1, 2, 3] }`":

```
let v := Vector.mk (Array.mk [1, 2, 3])
```

What the compiler means to say is "you need to provide a value for the second argument of the `Vector` constructor". Adding `rfl` as the second parameter resolves the error.

In some IDEs, system feedback is real-time, i.e., provided as soon as the user finishes typing. Our participants were divided on whether this was a good thing. If someone was working in an on-demand IDE (such as Emacs with keyboard shortcuts), they also preferred on-demand feedback, whereas participants working in real-time IDEs (such as VS Code with continuous checking) preferred real-time feedback. Participants who preferred on-demand feedback criticised real-time feedback for its tendency to interrupt them while they were still thinking. On-demand feedback proponents argued that being interrupted helped them stop and think before going down a potentially infeasible path. Some even mentioned they would prefer a feature where the system would try to falsify their claims automatically, warning them before they would attempt to prove a false statement. These participants had previously seen this feature in Isabelle [32], an interactive theorem prover not based on dependent types.

[P8] I think they'll get that eventually, Isabelle's ability to tell you that a theorem is unprovable using something like QuickCheck automatically. And the automatic part is interesting. Like this kind of proof assistant point of view, like I'm really trying to help you as a user as you go. And if you make a typo in a theorem statement, you know, like two variables that should be the same or different names, and it's just obviously not satisfiable, then just tell me before I even start coding.

H6: Effectively responding to the DTPL's requests requires experience.

[P8] As a user of Lean, I'm not bothered. I find error messages good enough. I mean, for example, the error message about universe is not matching. I wasn't looking closely at it, but it gave me the information [that] something [was] wrong with levels. That was fine.

Our experienced participants could, in most cases, "translate" error messages into concrete requests for information, but the beginners had a lot more trouble interpreting them. They understood the proof assistant wanted *something*, but had difficulty pinpointing the problematic part of the code.

[P4] I was using AI tools basically to explain to me the errors because I don't understand the errors.

3.4 Control

H7: DTPL users want the entire system to behave more like they think.

In general, the participants were positive about the experience DTPLs provide. One common source of frustration, however, was having to "over-explain" reasoning steps. Frequently, the programmer could clearly see the steps ahead that would get them to their goal, but the DTPL did not. When this happens, the programmer has to manually type out the steps, which feels tedious.

[P3] But then at some point I got this problem with [data types]. I was using pattern matching on a let bound thing, which was not a record, like, oh, man, there's only one constructor. I can either change it to a record or, you know, fiddle around a little bit with stuff. And that was a distraction in the development, I feel.

One such source of frustration is related to the unfolding of definitions. When a function is applied to known arguments, the resulting term can potentially be evaluated further by inlining the function definition and β-reduction. This can be useful, potentially leading to simpler goals, but can also make it harder to recognise the remaining sub-goal or identify which lemmas can be used to complete the proof. Several participants expressed their wish to have control over when and where unfolding is applied.

> [P2] […] you really want to be careful about where you unfold and how you unfold things. Sometimes it's really useful to be able to see the non-normalised goals, [sometimes] to see the fully normalised goal.

4 Recommendations

During our conversations with the participants, we noticed that there was some kind of consensus about certain DTPLs being good for certain niche tasks. For projects involving complex maths, Lean is the most obvious choice, given the existence of MathLib and custom proof automation. Exploring type theory or programming with dependent types is easier in Agda. The creation of a verified program, such as a compiler or a piece of embedded software, is more often done in Rocq. It is important to emphasise that, theoretically, the DTPLs are interchangeable, but depending on the task at hand, one DTPL might provide a smoother experience than another. Additionally, most participants indicated they do not see much use in using their DTPL of choice outside their particular niche.

> [P1] […] users that do use it, I think, like it a lot because it's for its specific purpose, and that's sort of a programming environment where you can work with dependent types.

In this section, we therefore discuss recommendations on how to make DTPLs accessible for a wider range of programmers in more mainstream contexts. We do not think that all DTPLs should strive towards this goal, since we saw that our participants valued how these DTPLs currently apply to their niche. Changes that would make a DTPL more appealing to a wider audience may also make it less appealing to its current user base. However, we believe there is value in making DTPL technology more accessible, and provide recommendations grounded in our data on what to focus on. We also highlight research that has already made progress in this direction.

R1: Keep it simple and design for inexperienced users first.

Learning to use a new programming language is not easy, and we identified several areas where we observed learners and experts struggling. Most of these obstacles are not unique to DTPLs, but the addition of dependent types both magnifies the obstacles for the user and adds an extra layer of complexity to the potential solutions. We zoom in on a few design considerations that were brought up during the interviews, but our overarching recommendation is to prioritise designing for new rather than expert users.

Installation. In preparation for the interviews, some participants had to reinstall their DTPL, which was not always a smooth process and many participants commented on it at the start of their interview[4]. For example, the Agda editor and standard library have to be installed separately. This lead to some participants having to conduct their interview without standard library support, much to their frustration. Rocq and Lean have comparatively straightforward installation instructions, but troubleshooting can be difficult for all of them. Participants would have liked a more straightforward installation process, preferably directly from an editor they are already familiar with.

Design of Standard Libraries. There is some tension in the design of standard libraries. On the one hand, they typically aim to be widely applicable and reusable but, often, the most general definition is not the easiest to understand. Consider, for example, the very general type signature of function composition in Agda's standard library:

```
∀ {A : Set a} {B : A → Set b}
    {C : {x : A} → B x → Set c}
    → (∀ {x} (y : B x) → C y)
    → (g : (x : A) → B x)
    → ((x : A) → C (g x))
```

This is the corresponding simply typed version:

```
∀ {A B C : Set} → (B → C) → (A → B) → (A → C)
```

Not only does the dependently typed version abstract over the universe levels involved, the functions being composed may themselves be dependently typed. Yet this type is hardly recognisable as function composition! During the interviews, participants complained about the effort needed to specialise a library function for their use case.

> [P2] I like the standard library so abstract because that means that it is applicable to many things, but it is also really painful sometimes too. Because everything is so abstract, you really have to put in work to apply [the functions] there to get the functionality that you want.

Participants programming in Lean, on the other hand, did not face these issues. One possible explanation we offer is that many of Lean's libraries are more tailored to the available language support, making use of type classes, for which Lean synthesises instances, and universe polymorphism, for which Lean has inference. Function composition in the Lean standard library, for example, does not include dependent types in its signature, but does abstract over the universe levels. This leads us to believe that the adoption of highly general definitions in a standard library requires targeted language support to be employed effectively.

Along a similar vein, the standard libraries are difficult to search through due to their architecture as well as the difficulty of recognising general type signatures as something applicable to the programmer's current context.

> [P3] I want to find the lemma that plus is commutative in the standard library. It's somewhere hidden in the fact that blah, blah, blah... There's like seven layers of records and padding around it where I think, oh, I shouldn't.

[4] Some of these comments are not caught on our recordings, because they were usually made directly upon joining the call.

Often, the only method of finding the definitions programmers are looking for is to search for the expected name of the lemma in an entire directory. Some libraries, such as the Lean standard library, offer a web page with search functionality, and there exist tactics that can search for all applicable proofs in the current context. Still, even participants working with well-documented libraries would have liked better searching to be available directly in the IDE. Additionally, two participants expressed their desire for a search by "lemma shape" (which Rocq already supports) a template of the theorem they are interested in (e.g., `IsEven _ -> divides 2 _` for "I am looking for some implication which can tell me something is divisible by two if I give it an even number"). We conclude that good in-IDE searching would contribute to making the entire system easier to use.

Syntax. Juhošová et al. [23] identifies *syntax* as a learning obstacle of Agda. In the end only the student participants complained about syntax issues and Unicode characters. This confirms that it is indeed a learning obstacle, but perhaps an obstacle that can be overcome.

> [P6] I think it's because [the syntax is] very, I don't want to say strict, but very specific. Even when writing this, because I forgot a space, then it was not compiling.

Nonetheless, this obstacle to learning should be taken seriously. The only participants who expressed an overall negative sentiment towards DTPLs were the students; we deem it unlikely they ever willingly choose to use a DTPL again. Furthermore, there was consensus between some expert and student participants: the usage of Unicode characters and specific notation is fine in contexts where their meaning is universally understood. Outside those contexts, however, highly specific custom notation obscures the intended meaning of proofs and programs. We therefore believe that syntax does not matter, except when it gets in the way of the learning experience, and effort should be made to report such syntax errors as clearly as possible.

Error Messages. Both students and experts complained about unclear error messages during their think-aloud sessions. Even experts were not always able to correctly interpret the error message and identify the relevant issue in their code. Furthermore, error messages are sometimes phrased using terms that are unfamiliar to beginners, for example, referring to meta-variables or unification errors. Even though we saw that it is possible for users to learn the compiler's jargon, these systems could be made more accessible by providing error messages with fewer technical terms, focusing on how to resolve the problem.

Relevant work in this direction is *type error diagnosis* [43], with techniques on identifying the most likely causes of a type error [19] and using counter-factual typing to suggest changes based on type errors [7, 13].

Working with Dependent Types. Programming and proving with dependent types sometimes exposes details about the underlying type theory, to which our participants directly attributed a class of problems we dubbed "representation problems". These representation problems share a common cause: the choice of data representation and function implementation has a significant impact on the development and proof effort — much more so than when doing pen and paper proofs. This problem was most prevalent among the mathematicians unfamiliar with dependent type theory:

> [P9] It uses dependent type theory and I'm a mathematician. I'm not that familiar with dependent type theory, and it's just things that are really obvious to me are non-trivial because the types are different.

Types in DTPLs are considered equal up to some notion of *conversion* that includes at least β-equality. As a result, different implementations of the same function may lead to different proofs. To illustrate this point, consider the usual definition of addition on natural numbers by induction over the first argument. The proofs that $0 + n = n$ and $n + 0 = n$ hold for all n are vastly different: the first holds trivially by evaluation; the second requires an inductive argument. Unsurprisingly, proofs involving *equality* were one of the greatest sources of such representation problems. Agda, Rocq, and Lean all have several notions of equality (including definitional equality, propositional equality, and setoid equality), and it is not always possible to convert between them.

Another common complaint we observed was related to how functionality is bundled. In many DTPLs, there is more than one way to do so: records and instance arguments in Agda; structures and type classes in Rocq and Lean. As a result, we observed frustration among the participants when combining different APIs that use different language features to model conceptually similar notions.

Although representation problems may seem inherent to DTPLs, we argue that it is worth exploring how they can be mitigated or at least minimised. The goal here is not to remove any trace of dependent types, since many participants found their guidance useful. On the contrary, we expect that the rich static type information that dependent types provide can be leveraged to help guide users. We have already seen research in this direction with *proof transport* [10] and control over unfolding [17].

R2: Prefer existing, "good" solutions over theoretical, "perfect" ones.

The design of DTPLs is a subtle affair. Each new DTPL typically explores new points in the design space This often causes editors (or plug-ins for existing editors), documentation generators, and other tooling to be designed from scratch. This is certainly appealing from a research perspective, but not always the best choice for usability. We recommend that DTPL maintainers consider compromising on

their design vision in order to produce more usable systems overall.

Many DTPLs are developed by researchers and lack substantial backing for engineering work and maintenance. This means that when a tool in the ecosystem has reached the "proof of concept" stage, it is not always possible to further develop and maintain it. While this might be enough for initial adoption, it leaves no space for additional improvements. Even though DTPLs are different from mainstream programming languages, they *are* still programming languages. This means that solutions to ecosystem requirements already exist. One example is the Language Server Protocol (LSP), which can be used to get support in multiple editors without having to tailor the solution to each supported editor. We realise that using existing tooling does not guarantee development will be easier, but the already available documentation, developer experience, and design considerations can be a great advantage. In the cases where having the "perfect" solution is not feasible, we recommend using an existing "good" solution, with can be improved and maintained by a wider range of developers.

Similarly, we recommend that the various DTPL communities learn from each other. Multiple participants complained about something being easy in one system, but close to impossible in another.

> [P2] Yeah, I mean, I started out with [Rocq] … and there's a lot of automation of tactics that can be used to automate things. I used to appreciate that about [Rocq] and I used to miss it a little bit when I was programming in Agda.

We have already seen success from such cross-fertilisation between different DTPLs. For example, the dependently typed pattern matching in Agda has inspired the Equations framework in Rocq [38, 39]. The work in Idris [5] on elaborator reflection [8] was ported to Agda, replacing the previous meta-programming features [42]. Venues, such as the *Workshop on Implementation of Type Systems* (WITS), enable developers of different systems to share their experience and best practices. We believe that expanding such collaboration and cross-fertilisation will lead to better systems in general.

R3: Make sure that users can rely on existing components and standard libraries to stay stable.

The purpose of a standard library is to reduce the amount of work programmers have to do by defining common types and functions for them. This is why we were surprised to learn some participants using Agda actively tried to avoid using the standard library, voicing their concerns about backwards compatibility.

> [P2] If you want to apply, well isomorphisms, I think there are like three different notions of isomorphisms … in the standard library and you should use a particular one. And that particular one has changed recently … and then all of the functions surrounding it changed.

If using a definition from the standard library means the programmer has to rewrite a portion of their code the next time they update their DTPL, one of two things will happen: the programmer will not update their DTPL, or the programmer will not use the standard library. Similar reasoning holds for other tooling. We recommend that library and tool maintainers put emphasis on remaining stable unless absolutely necessary, and maintain clear communication about breaking changes.

R4: Help new users join the community and understand the relevance of your system.

All three DTPLs already have active communities, and almost all our participants were positive about them. As a result, we expect these DTPLs to become better over time through contributions from their community members. We do, however, believe that larger, more diverse communities can help DTPLs improve faster, and recommend focusing on *accessibility* and *education*.

Accessibility. The beginner participants all encountered at least one problem they were not able to solve. Some immediately opted to ask an AI assistant, but some attempted to understand the error message or consult the official documentation, which often proved to be a fruitless endeavour. This suggests that there is a deeper issue around the *accessibility* of these systems. The error messages, documentation, and libraries typically target fellow experts, rather than beginners. Similarly, contributing to DTPL tools can be challenging, since the repositories often lack READMEs and contribution guides, and the code is only sparsely commented and has a non-standard or under-documented architecture. Improving the overall accessibility of these resources may help grow and diversify the user base.

Education. Our other recommendation for community growth is for teachers to provide compelling use cases that illustrate the benefits of these systems. Since our beginner participants were students who came across Agda in a course within their Computer Science bachelor study, we expected them to have a good idea of how they could apply DTPLs in potential future careers as software engineers. This proved to not be the case, and many of them struggled to grasp how these systems could be used in practice. We consider helping potential new users understand the relevance and value of DTPLs a crucial step towards expanding the community.

R5: Provide a *native* and *accepted* way to opt out of totality.

Developing software in a dependently typed language, even without writing any proofs, can take substantially more time and effort than it would in a simply typed programming language. One of the main factors contributing to this overhead is the fact that DTPLs typically require all functions

to be *total*[5] to ensure that the underlying logic is sound. Yet, any real world application is full of partial functions, such as parsing input data, array indexing, integer division, or numerous other (effectful) computations that may fail dynamically. In many cases, these *can* be made total easily enough, for example, by explicitly working with the `Maybe` type or by passing preconditions as additional arguments. Each of these solutions, however, comes at a very real development cost: callers of these partial functions must handle exceptions or prove that they cannot occur.

Many projects in DTPLs focus on working on the subset of the problem domain where all functions are total, with the rest usually being delegated to another language. Functions written in Rocq or Agda can be extracted to languages (e.g., OCaml, Scheme, or Haskell) [9, 27, 41], where invoking partial or effectful functions is more commonplace. However, this approach was not very popular among our participants, with one showing scepticism over its usability:

> [P1] Like, I think there's [a] perceived potential design pattern where you write code in Agda and you verify it, and you extract the sort of executable part. … you write an untyped unsafe program, and then you prove (after the fact) that it's safe. It's, I think, the wrong way to do it in a dependently typed language. It's like really your invariants and your code should be tightly merged.

We do not want to discredit the potential effectiveness of code extraction, however, extraction introduces non-trivial overhead required to integrate code in two languages. DTPL maintainers wishing to provide seamless code-extraction features need to spend time integrating their tooling with existing tooling for the target language. Users need to learn to work in both languages, where context switching can introduce unnecessary additional cognitive load. We worry that this workload it is too high for widespread adoption. This is why we argue for *native* support for totality opt-outs, so DTPLs can interact more seemlessly with other languages.

Existing DTPLs do provide some support for defining partial functions. For example, Agda and Lean allow developers to explicitly mark certain functions as terminating, overriding the termination checker, but potentially compromising on the soundness of their solution, but as the quote above suggests, they do not sit well with some users. The risk of scrutiny from colleagues and reviewers leads DTPL users to avoid such features. We stress that the challenge here is not only technical: such an opt-out needs be *accepted* by the community to become effective.

We suggest looking towards more mainstream languages which have succeeded in resolving similar tensions between rigour and usability. One approach would be to classify programs into two categories: a verified core and an unverified outer shell, similar to how Haskell [18] encourages writing a pure core and encapsulating all effects in the `IO` monad. Alternatively, partial functions may explicitly be given an exception — much in the way Rust [25] allows code to be marked as unsafe[6]. In both of these cases, the approaches are integrated into the language. The community has accepted guidelines on when and where these "escape hatches" are permissible, and they are presented as features of the respective language in the official documentation.

5 Threats to Validity & Study Limitations

Researcher Bias. We have applied the Socio-Technical Grounded Theory methodology to develop theories grounded in qualitative data, and used techniques which inherently integrate our culture, identity, and experiences into the analysis. We acknowledge the constructivist nature of our research paradigm, and ensure the soundness of our work by following well-established guidelines. Based on Hoda's guide to STGT [21, p. 260], our research team regularly reviewed the codebook used during the analysis and discussed the emerging concepts and categories during weekly meetings. Furthermore, as recommended by Hoda [21, pp. 338–339], we provide a detailed overview of how we applied STGT in Section 2, including a description of how participants were recruited, an explanation of how data was collected and analysed, and a demonstration of our coding process with specific examples. We provide our entire codebook as well as the interview transcripts in our public data repository [4] under an MIT Licence. Finally, when presenting the results in Section 3, we "present a clear chain of evidence from [participant] quotations to proposed concepts" [1], as prescribed by the *Empirical Standards for Software Engineering Research*.

Recruitment. We used a combination of convenience sampling and online sign-up to recruit our participants (see Table 1 for the division). While an attempt was made to spread the sign-up survey as wide as possible, its reach was still influenced by our environment. This way of sampling is what lead to the majority of participants being Agda users, which also influenced the direction we took during the STGT iterations. Though we believe to have reached theoretical saturation for the concepts and categories we focused on in our data, a different recruitment strategy might lead researchers towards different avenues of interest.

Interviews. Our dataset consisted entirely of qualitative data in the form of interview transcripts. While this was a suitable way to "find the most […] possible values of a characteristic in the population", it does mean we are unable to "identify the distribution of characteristic values" in that population [30, p. 1]. Our findings leave us with hypotheses that are grounded in our dataset, but future research which applies a combination of qualitative and quantitative

[5] A total function requires every input from the domain to produce an output in the codomain.

[6] Similar to how Agda allows functions to be explicitly marked as terminating when the termination checker fails to prove it.

techniques could help refine them. For example, it would be interesting to study the prevalence and severity of the positive and negative experiences reported in our work.

Diversity. Only two of our participants were not men, and neither was an expert in using DTPLs. While we do have the impression that this is a representative sample of the *current* DTPL user population, we believe a more diverse participant pool would have provided us with richer insights. We strongly encourage future research to target a *potential* population instead, if only to determine how best to adjust the object under study to cater to it.

6 Related Work

Though dependently typed programming languages have been around for some time, user-oriented research has only recently picked up in the field. We discuss the most relevant studies to our work below.

Similarly to us, Shi et al. [36] conducted think-aloud session with their participants, followed by a short interview in which questions were based on the observations from the think-aloud. Though they focused specifically on proof writing in Rocq and Lean (both tactic-oriented DTPLs), their findings closely reflect ours. They found that their participants often consulted external resources, interacted with the compiler and advanced by incorporating its feedback (H5), considered design aspects beyond getting to a correct proof and refactored their proofs to match a certain style even after it was finished (H4), and struggled with "minutiae" when trying to convince the compiler that they are satisfying the constraints laid down by the types (H7).

Lubin and Chasins [28], whose Grounded Theory study focused on the usage of static type systems, presented hypotheses that are reflected in our own findings. Our observation that refactoring types is a necessary part of writing a DTPL program (H1) is in line with their first hypothesis stating that functional programmers iterate between editing types and editing expressions. Their second hypothesis, stating that functional programmers run their compilers on code they know will not compile, fits into our theory about conversing with the proof assistant (H5).

Juhošová et al. [24] found that advanced typed systems help users plan and guide their implementation, and allow them to hold an active conversation with the compiler. These findings correspond with our hypotheses on planning (H1) and interactivity (H5). Additionally, their findings about the obstacles that new users face [23] and the improvements that experienced users would like to see [24] support the reasoning behind our recommendations: they find that the coupling between dependent type theory and the language design is a learning obstacle (R1) and that the lack of support for writing real-world applications makes it feel "irrelevant" (R4). A similar study on the usability barriers with liquid types was conducted by Gamboa et al. [14]. While systems with liquid types work differently to DTPLs, their findings are consistent with ours. Specifically, they also identify limited IDE support, lack of reference resources, complex setup, and unhelpful error messages as usability barriers.

7 Conclusion & Future Work

In this work, we used Socio-Technical Grounded Theory to investigate how people program in dependently typed languages. Based on twelve interviews with users of DTPLs, we defined seven hypotheses and described five concrete recommendations for anyone who wishes to make DTPL technology accessible to a wider range of software developers. These recommendations currently specify *what* to focus on, but not necessarily *how* to achieve the desired results. Promising avenues of future research include investigating how to make in-IDE searching of proof libraries better; how error messages can be simplified and made more actionable; and how to better hide the complexities arising from a DTPL's underlying type theory from the user. Additionally, if the goal is to provide everyday developers with access to DTPLs, we believe it is important to investigate how to do so while keeping the solution *practical*, and to consider how DTPLs as a tool fit into the wider context of software engineering. Crucially, we think it is important that any future research in this direction is conducted in a user-oriented rather than theory-driven manner — tools designed for developers should put developer experience first. We hope that this work helps guide the future development of DTPLs, eventually enabling a wider range of developers to write both *programs and proofs*.

Data-Availability Statement

We provide a MIT-Licenced data repository containing resources for the interview sessions we conducted, anonymised transcripts of those interviews, and our codebook [4].

References

[1] 2020. Empirical Standards for Software Engineering Research. https://arxiv.org/abs/2010.03525v2

[2] Steve Adolph, Wendy Hall, and Philippe Kruchten. 2011. Using grounded theory to study the experience of software development. *Empirical Software Engineering* 16, 4 (2011), 487–513. https://doi.org/10.1007/s10664-010-9152-6

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer, 50–65. https://doi.org/10.1007/11541868_4

[4] Ruben Backx, Sára Juhošová, and Wouter Swierstra. 2026. Dataset for: "Programs and Proofs in Practice: A Grounded Theory on How People Program with Dependent Types". https://doi.org/10.4121/1ea19b98-3007-4f36-8216-e05e36c3d1db

[5] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Co.

[6] Kathy Charmaz. 2014. *Constructing Grounded Theory* (2 ed.). Sage Publications.

[7] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *Symposium on Principles of Programming Languages (POPL '14)*. ACM, 583–594. https://doi.org/10.1145/2535838.2535863

[8] David Christiansen and Edwin Brady. 2016. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, 284–297. https://doi.org/10.1145/2951913.2951932

[9] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. 2022. Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs. In *International Haskell Symposium*. ACM, 108–122. https://doi.org/10.1145/3546189.3549920

[10] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Trocq: Proof Transfer for Free, With or Without Univalence. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer, 239–268. https://doi.org/10.1007/978-3-031-57262-3_10

[11] Juliet Corbin and Anselm Strauss. 2015. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (4 ed.). https://uk.sagepub.com/en-gb/eur/basics-of-qualitative-research/book235578

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

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

[14] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3729327

[15] Barney G. Glaser. 1978. *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Press.

[16] Barney G. Glaser and Anselm L. Strauss. 1967. *The discovery of grounded theory: strategies for qualitative research*. Aldine Publishing, Chicago. OCLC: 253912.

[17] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2025. Controlling unfolding in type theory. *Mathematical Structures in Computer Science* 35 (2025), e38. https://doi.org/10.1017/S0960129525100327

[18] Haskell.org. [n. d.]. Haskell Language. https://www.haskell.org/

[19] B. J. Heeren. 2005. *Top quality type error Messages*. Ph. D. Dissertation. Utrecht University. https://dspace.library.uu.nl/handle/1874/7297 Accepted: 2005-09-20T12:43:01Z.

[20] Rashina Hoda. 2022. Socio-Technical Grounded Theory for Software Engineering. *Transactions on Software Engineering* 48, 10 (2022), 3808–3832. https://doi.org/10.1109/TSE.2021.3106280 Conference Name: IEEE Transactions on Software Engineering.

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

[22] Rashina Hoda, James Noble, and Stuart Marshall. 2012. Developing a grounded theory to explain the practices of self-organizing Agile teams. *Empirical Software Engineering* 17, 6 (2012), 609–639. https://doi.org/10.1007/s10664-011-9161-0

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

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

[25] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language* (second ed.). No Starch Press, San Francisco, CA, USA.

[26] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[27] Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*. Springer, Berlin, Heidelberg, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[28] Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 155:1–155:30. https://doi.org/10.1145/3485532

[29] Zainab Masood, Rashina Hoda, and Kelly Blincoe. 2022. Real World Scrum A Grounded Theory of Variations in Practice. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1579–1591. https://doi.org/10.1109/TSE.2020.3025317 Conference Name: IEEE Transactions on Software Engineering.

[30] Jorge Melegati, Kieran Conboy, and Daniel Graziotin. 2024. Qualitative Surveys in Software Engineering Research: Definition, Critical Review, and Guidelines. *IEEE Transactions on Software Engineering* 50, 12 (2024), 3172–3187. https://doi.org/10.1109/TSE.2024.3474173

[31] Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. 2025. On the Impact of Formal Verification on Software Development. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 403:3642–403:3668. https://doi.org/10.1145/3763181

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

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

[34] Aastha Pant, Rashina Hoda, Chakkrit Tantithamthavorn, and Burak Turhan. 2024. Ethics in AI through the Practitioner's View: A Grounded Theory Literature Review. https://doi.org/10.48550/arXiv.2206.09514 arXiv:2206.09514 [cs].

[35] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust Speech Recognition via Large-Scale Weak Supervision. https://doi.org/10.48550/arXiv.2212.04356 arXiv:2212.04356 [eess].

[36] Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2025. QED in Context: An Observation Study of Proof Assistant Users. *Artifact for QED in Context: An Observation Study of Proof Assistant Users* 9, OOPSLA1 (2025), 92:337–92:363. https://doi.org/10.1145/3720426

[37] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. Identifying design problems in the source code: a grounded theory. In *International Conference on Software Engineering (ICSE) (ICSE '18)*. ACM, 921–931. https://doi.org/10.1145/3180155.3180239

[38] Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 419–434. https://doi.org/10.1007/978-3-642-14052-5_29

[39] Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Equations Reloaded Accompanying Material* 3, ICFP (2019), 86:1–86:29. https://doi.org/10.1145/3341690

[40] The Coq Development Team. 2024. The Coq Proof Assistant. https://zenodo.org/records/14542673

[41] The Coq Team. [n. d.]. Extraction of programs in OCaml and Haskell. https://coq.inria.fr/doc/V8.11.1/refman/addendum/extraction.html

[42] Paul van der Walt and Wouter Swierstra. 2013. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer, 157–173. https://doi.org/10.1007/978-3-642-41582-1_10

[43] Mitchell Wand. 1986. Finding the source of type errors. In *Symposium on Principles of programming languages (POPL '86)*. ACM, 38–43. https://doi.org/10.1145/512644.512648

[44] Michael Waterman, James Noble, and George Allan. 2015. How Much Up-Front? A Grounded theory of Agile Architecture. In *International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 347–357. https://doi.org/10.1109/ICSE.2015.54