The Way of Types: A Report on Developer Experience with Type-Driven Development

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

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

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

ABSTRACT

Static type systems are a well-established tool for preventing basic software errors, with more advanced ones providing strong guarantees of program correctness. Additionally, type systems encourage a "types first, implementation later" developer workflow known as "type-driven development" (TyDD). However, current widespread TyDD practices are based on simple type systems with limited expressivity, and the advanced tools being developed by researchers are not making it into mainstream programming languages. To determine *how current practitioners experience the use of type-driven development* and *what inhibits its adoption by a wider range of developers*, we conducted a survey with 130 participants from various backgrounds, asking them to describe their experience with current TyDD tools. According to them, TyDD can *guide*, *communicate*, and *verify* program implementation, but is currently limited by usability issues and missing features. Based on these results, we recommend that advanced TyDD tools be made available to a wider range of developers by investigating and addressing these limitations, with a focus on increasing expressivity while preserving usability.

1 INTRODUCTION

Static type systems are one of the most commonly used tools for *formally* verifying software. Unlike other established verification methods such as software testing [2] or code review [4], static type systems can be used to completely prevent certain classes of errors at compile-time. As Dijkstra put it, "program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence" [13, p. 864]. Static type systems, on the other hand, are a method for "proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute" [41, p. 1]. For example, they can enforce simple constraints such as "retrieving the element at the i^{th} index can only be performed on a list".

The picture of type systems as purely a tool for error prevention is, however, incomplete. An important object of study within programming language research is how types influence program-writing, via an approach often referred to as "type-driven development" (TyDD). In his textbook on TyDD, Brady describes this approach as involving "[writing] types first and [using] those types to guide the definition of functions" [6, p. 5]. For example, he defines a *type-driven* approach to reasoning about message-passing concurrent programs as "writing an explicit type describing the pattern of communication, and verifying that processes follow that pattern by type checking" [7, p. 2/22]. Brady claims that the name "type-driven development" suggests an analogy to test-driven development [44]: in both cases, you "first establish a program's purpose and whether it satisfies some basic requirements" [6, p. 3] and only then implement it.

We can already see basic forms of type-driven development in current mainstream programming languages. For example, writing an interface in Java is essentially first expressing some desired behaviour using types and only then implementing it. Over the past decades, programming language researchers have designed and investigated many advanced type systems, including

- *ownership types*, used in Rust [27], which enforce safe memory usage in concurrent settings and in the absence of garbage collection;
- *dependent types*, used in theorem provers such as Agda [39], Rocq [50], or Lean [12], which can enforce constraints such as "retrieving the element at the i^{th} index can only be performed on a list with *more than i elements*", but are also used to formalise and prove mathematical theorems; and
- *effect systems*, which can be used to describe program side effects and check for their occurrence at compile-time [38].

These advanced type systems come with powerful derived tools which facilitate type-driven development such as typed holes (see Section 2.2) or type-based library search such as Hoogle [37]. Unfortunately, these tools currently live in relatively niche languages, often as prototypes or proofs-of-concept, and are not available in widespread programming language ecosystems[1].

Anecdotally, software engineers who are used to working with static types systems often speak of "missing them" when using a programming language without one. Similarly, they prefer using languages with static types when working on large-scale, collaborative projects which prioritise reliability. Even Python and JavaScript, both popular programming languages designed to be dynamic and flexible, have statically-typed versions in the form of `mypy` and TypeScript respectively. Though these static checks can be circumvented, and the code can be executed regardless of whether the type checking succeeds, it is considered bad practice to do so when working on industry-standard software.

So what exactly is it that practitioners "miss" when they do not have static types available? And why have we not seen too many advanced type systems make their way into mainstream programming language processes? We imagine that type-driven development could become an even more valuable tool for software engineering, if we can pinpoint the current developer experience

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

and improve it with the best practices from the software engineering community. To that end, this work aims to achieve two goals via three research questions:

Goal 1: Understand how current practitioners experience the use of type-driven development.

RQ1.1: *What does the practising community understand under the term "type-driven development"?* Though we have definitions of TyDD from Brady [6, 7], we wish to see how the approach is perceived in practice.

RQ1.2: *Which perceived benefits do practitioners gain from using type-driven development?* This research question aims to uncover the aspects of TyDD that are actually perceived as valuable by current practitioners — and perhaps show why making advanced TyDD tools available to a wider range of developers is a worthwhile pursuit.

Goal 2: Identify what inhibits the adoption of advanced type-driven development by a wider range of developers.

RQ2: *Which improvements would practitioners like to see in existing tools facilitating type-driven development?* Understanding what practitioners currently struggle with when applying TyDD might give us some indicators as to why its more advanced tools have not been adopted into mainstream developer processes.

Based on an online survey with 130 practitioners, we conclude that type-driven development is considered to be an approach to program-writing which helps *guide*, *verify*, and *communicate* the implementation. Practitioners feel that it supports their mental model of the problem and implementation, gives them confidence in their solutions, and makes the overall development process more enjoyable. Based on the improvements our participants wished to see, we identified three inhibitors to the widespread use of current advanced TyDD tools: (1) they do not sufficiently shield users from their underlying complexities, (2) the quality of their ecosystems is inadequate, and (3) they miss library and tool support for building applications which can interact with the real world. As a result, we suggest a usability-oriented research agenda for gradually increasing type expressivity within mainstream programming languages.

2 TYPE-DRIVEN DEVELOPMENT

Consider the following pseudocode, which defines a recursive implementation for getting an element at a certain position in a list:

```
fun get(index, list) {
  if (list.isEmpty() || index < 0) return null
  if (index == 0) return list.first()
  return get(index - 1, list.subList(1))
}

print(get(['a', 'b', 'c'], 2))
```

If we were to actually run this code, we would get a runtime error: *"cannot call* `isEmpty()` *on the number* `list` *(line 2)"*. The interpreter has *detected* an error: we are calling the function with our arguments in the wrong order. We can now apply the simple fix and move on. Unfortunately, there is nothing stopping us from repeating this error somewhere else and no guarantee that we will be able to *detect* it before it causes trouble.

With the help of a *static type system*, we can *prevent* this class of errors altogether. With the function signature type-annotated, our adjusted code snippet would throw the following error during compilation, i.e., before even running the code:

```
fun get<A>(index: Int, list: List<A>): A? {..}

print(get(['a', 'b', 'c'], 2))
```

```
type mismatch for argument index (line 3):
  expected:    Int
  actual:      List<Char>
```

We are now communicating two crucial things to the caller of our `get` function: (1) that we expect an integer and a list of elements from them in that order, and (2) that they should expect to receive a value of the same type as the elements in the list *or null*.

The following two subsections describe how we can go further with type-driven development by encoding more expressive properties in the type signatures and interacting with the compiler in order to find an implementation.

2.1 Encoding Properties in Types

With a *dependently-typed system*, we can encode even more complex functional properties into the program's type signatures. In the following snippet, we "index" the types of the arguments with the natural number `n` — the types now *depend* on it:

- `Int[0,n]` means that we expect an integer between `0` (inclusive) and `n` (exclusive), and
- `List<A>[n]` means that we expect a list *of length* `n`, containing elements of type `A`.

Because `n` is defined as a natural number instead of an integer, we know that it will always be a valid upper bound and list size.

```
fun get<n: Nat, A: Type>(
  index: Int[0,n],
  list: List<A>[n]
): A {
  if (index == 0) return list.first()
  return get(index - 1, list.subList(1))
}

print(get(4, ['a', 'b', 'c']))
```

```
cannot resolve constraints on n (line 9):
  in expression 4:                 n > 4
  in expression ['a', 'b', 'c']:   n = 3
```

The type checker will now throw a compile-time error if the provided index is out of bounds. Additionally, neither condition which would have caused the function to return *null* will ever occur:

`index < 0` is eliminated by the lower bound on the type, and
`list.isEmpty()` would be true if `n = 0`, but that would mean that `index` must be between `0` (inclusive) and `0` (exclusive) — which is impossible.

At this point, it is probably good to reflect on *why this matters* — are we not essentially just moving the same checks from the runtime to the compile-time? Well, firstly, encountering a runtime error in our untyped example is the best-case scenario: in some cases, the program would *not* error and simply behave in an undefined way, resulting in a bogus computation. Secondly, such a check can easily be lost in huge codebases, especially with many developers collaborating on it. A static encoding of the data's validity into the structure holding it, however, enforces validation at the entry point and makes it *persistent* in the rest of the program [26].

Unfortunately, it is not always a completely straightforward task to convince the type checker that you are, indeed, honouring all the contracts defined in the type signatures. In our `get` function example, the type checker would expect a formal proof of the array index bounds at every call site. Providing these proofs either requires a careful setup of the types to make the information about index bounds flow to the right place, or else requires manual intervention by the programmer to provide the proof terms explicitly. Either way, the hassle of having to encode these proofs and maintain them in evolving code often outweighs the desire for formal correctness in day to day software engineering.

An example of an attempt to reduce this effort are *liquid types* [52], which limit constraint expressivity so that the resulting problems can be automatically proven by an SMT[2] solver [11].

2.2 Interactive Programming with Types

An additional benefit of static type systems is their power to provide immediate feedback to the developer. *Typed holes* are a signature TyDD tool, allowing developers to put a "hole" in their code as a placeholder for some expression and to find a suitable implementation for it by interacting with the compiler. For example, we could begin with the following partial definition of a function that applies a transformation to each element in a list:

```
fun map<n: Nat, A: Type, B: Type>(
    f: A -> B,
    list: List<A>[n]
): List<B>[n] {
    if (list.isEmpty()) return [___]1
    return [___]2
}
```

The development environment would then display the following:

Goal 1: `List<B>[0]`
Goal 2: `List<B>[n]`

In the case of the first goal, the tooling can simply fill in the hole automatically — there is only one way to construct a list of length `0`. To find an implementation for the second goal, we could ask the tooling to suggest a way to construct the desired value. In the case of most functional programming languages, the tooling would probably refine that hole into two new holes, expecting a value prepended to a new list[3]:

```
fun map<n: Nat, A: Type, B: Type>(
    f: A -> B,
    list: List<A>[n]
): List<B>[n] {
    if (list.isEmpty()) return []
    return [___]1 :: [___]2
}
```

Goal 1: `B`
Goal 2: `List<B>[n - 1]`

Upon examining the first of these new holes, the tooling would present us with the following overview of the context:

Context:
`f : A -> B`
`List.first : List<A>[s > 0] -> A`
Goal: `B`

Since our goal is to build a value of type `B`, we probably want to apply function `f` to some value of type `A`. To preserve the order of the original list, this should be the first element of the input list. According to our context, retrieving the first element of the list requires the length of the list to be larger than `0` — which we know is true! We can now successfully apply the functions to obtain a value of type `B`. Following the same technique for the second hole, we end up with the following recursive definition for the function:

```
fun map<n: Nat, A: Type, B: Type>(
    f: A -> B,
    list: List<A>[n]
): List<B>[n] {
    if (list.isEmpty()) return []
    return f(list.first())
        :: map(f, list.subList(1))
}
```

In a real-world scenario, the context of a hole would show the components that are currently available in the scope. This means that without any additional filtering or sorting (provided by, e.g., the editor), the developer might have to search through a long list for what they need. Additionally, the `List.first` function might not be displayed unless it was explicitly imported into the scope, making it the developer's responsibility to know of the existence of such a function or to find one suitable for their needs.

3 STUDY SETUP

To answer our research questions, we conducted an online survey among current practitioners of type-driven development. In order to not influence the participants' answers with our preconceived ideas, most questions in the survey were open-ended and, as such, required qualitative research techniques to be analysed. This worked well with our goal to "find the most [...] possible values of a characteristic in the population" rather than to "identify the distribution of characteristic values" [33, p. 1]. This section details how we applied these qualitative techniques. The survey and responses are available in our data repository under an MIT license [24]. This study was conducted with the approval of the Human Research Ethics Committee at the Delft University of Technology.

[2]"Satisfiability modulo theories (SMT)" are used to determine the satisfiability of mathematical formulas.
[3]Lists in functional programming languages are defined recursively: they are either an empty list or an element prepended to a list. To construct a list with two elements ('A' and 'B'), we would write `'A' :: ('B' :: [])`, i.e., element `A` prepended to a list where element `B` is prepended to the empty list.

3.1 Survey

All data was collected through an online *anonymous* survey, distributed over the following channels, targetting mainly participants from communities that actively use *advanced* TyDD tools:

- the official Zulip chats of Agda, Rocq, and Lean[4],
- the official Zulip chat of the EuroProofNet[5],
- the official Discourse forums of Rocq and Haskell[6], and
- posts on social media by the authors.

The survey was available for a period of one month and could be filled in at each participant's own pace. Participation was voluntary, and respondents were free to skip any questions. The full survey is available in our replication package [24].

Demographics. We asked participants to fill in which role they identify with most as well as to select languages they have used for type-driven development. In order to include as many current users of TyDD tools as possible, we provided a wide range of roles to select from, including mathematicians using them to formalise their work and formal methods engineers using them to specify and verify software systems:

○ `mathematician`	○ `computer science researcher`
○ `mathematics student`	○ `computer science student`
○ `formal methods engineer`	○ `hobbyist`
○ `software engineer`	○ `other`

We offered a choice of languages with advanced or interesting type systems, and additional free-text fields for others they wished to mention. For each of their selected languages, we asked participants to indicate how they would rate their proficiency with it on a scale of `0` (Complete Beginner) to `10` (Expert) and to select the contexts they have used it in:

☐ `At my job in industry`	☐ `During a course / workshop / summer school`
☐ `For research`	☐ `For hobby projects`
☐ `For teaching`	

Questions. The main body of the survey was made up of open-ended questions about type-driven development. This paper presents results based on the questions that asked participants to:

(1) define what "type-driven development" means to them,
(2) list up to *three* benefits they think TyDD provides, and
(3) list up to *five* things they would like to see improved in existing TyDD tools.

In our cross-sectional study on the learning obstacles of an interactive theorem prover [25], we found that using short, free-text fields offers enough structure to receive the desired granularity of responses but does not restrict the participants to a set of preconceived ones. A similar approach has been taken by Robillard and DeLine in their field study of API learning obstacles [42]. From now on, we write "benefit / improvement entry" when referring to responses to one of the three or five individual fields of the respective question.

[4] `agda.zulipchat.com` `rocq-prover.zulipchat.com` `leanprover.zulipchat.com`
[5] `epn.zulipchat.com` `europroofnet.github.io`
[6] `discourse.rocq-prover.org` `discourse.haskell.org`

3.2 Pilot Study

To verify our survey design, we conducted a pilot study with five participants who had at least some experience with type-driven development. They were given access to an online preview of the survey, and asked to fill in the answers as they saw fit. Based on their responses and the feedback they provided in a subsequent conversation, we concluded that the general, open-ended structure of the survey was suitable, and only changed a few details about the wording of the questions.

Since the survey did not significantly change after the pilot, we included the initial participants' responses in our results.

3.3 Analysis

In order to analyse the free-text data, we applied a series of qualitative research techniques to each set of responses. Initially, the first author engaged in descriptive coding [36, pp. 55–69] of the submissions in order to "closely [inspect], deeply [make] sense of, and [infer] meaning from [the] data" [22, p. 226]. She assigned a single label (i.e. "code") to each benefit and improvement entry, since they each represented exactly one idea or concept. For example, both of the following benefit entries received the label `confidence in solutions`:

- "If it types, it works" [P163]
- "Peace of mind, about everything being correct or obviously wrong." [P218]

The TyDD definitions, on the other hand, often expressed multiple ideas and concepts, and thus usually received multiple labels. To illustrate, the following TyDD definition received four labels:

"Using the type system of a language to
constrain the possible implementations of a specification [1].
I closely associate type-driven development with
the practice of writing a type signature [2] and
an implementation with some holes [3], then
recursively refining those holes [4]." [P52]

[1] `minimising solution space`
[2] `types first`
[3] `interactive development`
[4] `interaction with compiler`

This resulted in one codebook per question. After the first iteration of coding, we looked at each set of responses per code separately, and recoded them as necessary.

Since the dataset was not overwhelmingly large, we did not have to introduce the complexity of multiple coders, thus achieving consistency without having to reach inter-rater agreement. We did, however, apply Hoda's technique for single analysts [22, p. 260]: we had regular meetings during which the codes were reviewed by the other researchers on the team and adjusted as necessary. We acknowledge the creative subjectivity of our codebooks (available in the data repository [24]) and declare our constructivist epistemological stance. To provide context for this stance, we specify that our team consisted of three researchers specialising in (1) dependent type systems, (2) software evolution and testing, and (3) the usability of advanced type systems.

Once we were satisfied with the codes, we created sticky notes for each one accompanied by short explanations, and sat together to

role	count
software engineer	54
computer science researcher	32
mathematician	14
computer science student	13
formal methods engineer	8
hobbyist	4
mathematician working as a software engineer	2
mathematics student	2
formal methods researcher	1

Table 1: Division of participants over role they identify with

diagram with them [8, pp. 218–221]. The resulting diagrams helped us understand the relationships between the ideas and concepts in the participants' responses and form the basis for our results in Section 4. Finally, we wrote memos — notes of "ideas about codes and their relationships as they strike the analyst" [20] — during our meetings and coding iterations and used them to discuss the significance of these results in Section 5.

4 RESULTS

The survey received a total of 281 responses [P0-280]. Unfortunately, 127 of these were completely blank, 24 contained only demographic information about the participants, and another 20 contained only the participant's definition for type-driven development. We decided to only include responses containing more than just demographics, resulting in a total of 130 processed responses. The data is available in our public data repository [24] under an MIT License.

The demographics of the processed participants are summarised in Tables 1 and 2. Almost 78% of our respondents associate with computer science, mostly as software engineers, but also as researchers or students in the field. The most commonly-used language was Haskell [32], a pure functional programming language, with 83% of participants claiming at least some experience with using it for type-driven development. The other two most popular languages were Agda [39], a dependently-typed language claimed to be used mostly for research and hobby projects, and Rust [28], a language with ownership types which has recently been rising in popularity.

4.1 Defining "Type-Driven Development"

Though we have a definition of "type-driven development" from Brady [6, 7], we wanted to see how the practising community understands the term. Based on the participants' responses, we were able to create the following definition:

> "Type-driven development is an approach to programming in which the developer defines a program's *types and type signatures first* in order to
> (1) *design* and (2) *communicate* the solution,
> (3) *guide* and (4) *verify* the implementation, and
> (5) *receive support* from related tools."

(1) Designing Solutions. The process begins with the developer "specifying *expressive* types" [P144] for the program they wish to write in order to "express the intent of the code before writing it" [P81]. This allows them to "focus clearly on the 'what' before [they] dive into the 'how'" [P72], essentially "modelling [the] domain" [P44] and "encoding business logic into types and transformations between them" [P57].

(2) Communicating Solutions. Type signatures "describe function inputs / outputs" [P65] and "specify the logic and contracts of your program" [P79]. They are used for "understanding library APIs as much [...] as the actual documentation" [P45], allowing developers to "think about how the different parts of the algorithm will come together later, without having to care about the actual implementation" [P63].

(3) Guiding Implementation. Once the types are defined, the programs "almost write themselves" [P15], directing your implementation and "reminding you to deal with all possibilities" [P15]. Type annotations "cut down the space of possible [implementations]" [P6] for a program to the point where it "can almost be derived from the signature" [P4]. Often, the "structure of the problem *drives* (even *determines*) the structure of the solution" [P18].

(4) Verifying Implementation. Checking implementations against their type signatures at compile-time makes "some erroneous states unrepresentable, and some classes of bugs impossible to write" [P40]. Developers "let the compiler catch [their] mistakes before [they] make them" [P115] making their programs "correct by construction" [P129]. By having the type checker "enforce critical invariants" [P47] before running the program, "[successful] compiling means that it is very likely correct" [P7].

(5) Tool Support. Many participants viewed the associated tooling and compiler support as an important part of the TyDD approach. They describe the "incremental" [P18] implementation process as "an active conversation between [them] and the compiler" [P35], where "the compiler will complain when [they] mess something up" [P45].

A core concept is using "holes" in programs (see Section 2.2), which can be filled "from top down with the correct implementations" [P76]. Tooling support for these usually includes context presentation (which type the hole expects and which variables are available), hole refinement (how the hole could potentially be split up into smaller holes), and even "deriving a correct implementation [for the hole] from types automatically" [P41].

4.2 Benefits

We received a total of 297 benefit entries from 108 unique participants, indicating what the benefits of TyDD are *as perceived by the participants*. Many of these point to and even overlap with the definitions of type-driven development from Section 4.1. According to the responses, type-driven development "gives structure in [the] development process" [P77] and even "provides a nice framework for [it]" [P37]. We identified ten main benefits that practitioners perceive to be gaining from TyDD:

(1) "Deeper understanding of the problem domain" [P44]. Because developers use complex types to "[express] exactly what [they] mean" [P12], they need to "focus on the conceptual understanding

		used for					proficiency	
programming language	**count**	**industry**	**research**	**teaching**	**learning**	**hobby**	**average**	**most common**
Haskell	106	42	43	22	24	73	6.5	8
Agda	64	6	36	13	21	33	5.4	3
Rust	58	20	18	3	4	43	5.8	7
Lean	49	6	26	14	11	29	5.2	2
Idris	34	3	11	1	4	26	4.3	2
TypeScript	32	24	6	2	3	9	6.7	9
OCaml	29	6	12	5	9	18	5.0	5
Scala	22	11	5	2	4	11	5.4	5
other statically-typed language	18	11	4	2	4	10	6.2	7
Dafny	8	1	3	1	1	6	4.1	5
a language with refinement types	8	0	6	0	1	4	6.0	7
Swift	7	2	1	2	0	4	5.4	2
other dependently-typed language	6	1	6	1	1	3	8.3	10
other language	6	4	3	2	2	3	7.7	10
Hazel	2	1	0	0	1	0	2.5	2

Table 2: Overview of participants' programming language experience

first" [P4]. This makes types "a good intermediate step between specification and operational code" [P10].

(2) More "thoughtful design" [P50]. TyDD "forces a problem to be more thought out before any implementation begins and may sometimes reveal flaws in potential structures" [P56]. It "nudges you to think and consider all your options" [P1].

(3) Easier mental models composed of interfaces. Because type signatures give you a clear interface for each component, TyDD "helps 'design at the level of interfaces', which leads to cleaner code" [P32]. It is "a mental tool to abstract away complex logic when reading / reviewing others' code" [P39], allowing you to "hold more of the program in your head when you consider just the signatures." [P211]. Additionally, it gives you the "ability to use a function or interface before its implementation is defined (e.g., in tests)" [P2].

(4) "Better collaboration through contracts and [APIs]" [P36]. Additionally, TyDD "[enables] collaboration based solely on interfaces" [P14], since "types can be easier discussed than code" [P176]. They "[expose a] 'hardened' API which is difficult for a misguided user to use incorrectly" [P33], meaning that library designers can trust the compiler to limit how their components can be used. It also enforces a certain "honesty about promises" [P28], telling users exactly which guarantees they can expect from a component.

(5) Maintainability of the resulting programs. TyDD facilitates "lower long-term maintainability costs" [P162]. Though we did not receive much context for these responses, we interpret them as relating to type-annotations as comprehension aids, which stay consistent and up-to-date in the face of software evolution thanks to being statically checked.

(6) Clearer path towards an implementation. "Since code is constrained by the types, there [are] usually only [a] few reasonable ways to write it" [P106]. Not only does this help with "[reminding developers] of corner cases that [they] might otherwise miss" [P15], it also offers possibilities to "automatically synthesise / generate the implementation" [P52].

(7) "Top-down development" [P58] via interactive development. Hole-driven, interactive development allows developers to "see the type of the goal and all variables in context" [P0], thus "easing the requirements on [their] short term memory" [P12]. Holes can also be broken up into multiple sub-goals, "each of which is easier to conceptualize and focus on" [P52].

(8) Higher confidence in the correctness of one's solution. As its main selling point, TyDD offers guarantees about the absence of certain bugs in the resulting programs. This allows developers to "[enforce] invariants that otherwise would only exist on paper" [P4], "feel confident that [their] systems work as [they] intended" [P72], and have "peace of mind about everything being correct or obviously wrong" [P218]. They can rely on the type checker to "inform [them] when [they] make mistakes" [P5] and to check that they are "not being stupid" [P12].

(9) "Way less scary" refactoring [P5]. Since types "allow one to make assumptions about [the program's] validity", they allow for confident optimisation [P112] and other refactoring. Type checkers catch "almost all forgot-that-spot mistakes when refactoring" [P69] and sometimes even allow developers to "turn most of [their] brain off without having fear of breaking something important" [P236].

(10) Pleasure when programming. Last, but definitely not least, participants find it "fun" to use TyDD — twelve entries listed this as a benefit. Participants mention that "it is relaxing when the later parts of coding require less focus" [P15], that they can "quickly [get] new success moments" [P266], and that it "makes [them] feel smarter" [P106].

4.3 Improvements

We received a total of 217 improvement entries from 72 different participants. Two of these entries were not included in our analysis: one was an incomplete sentence whose meaning we could not gauge, the other an opinion that we should no longer teach programming languages without type systems. We were able to identify four general areas of improvement for tools that facilitate type-driven development:

(1) shielding the users from the underlying complexities,
(2) improving a wide range of ecosystem features,
(3) integrating TyDD tools into the real world, and
(4) adding more features to existing tools.

(1) Shielding Users from Complexities. According to the responses, working in languages with strong type systems can be "complicated" [P50]. The main difficulties lie in the fact that

- "the learning curve is... very curvy" [P30],
- "debugging is cumbersome [since] sometimes code simply does not run because of some errors that a weakly-typed language might have ignored" [P50],
- "the stronger the type system, the more difficult it is to make the type checker understand what you mean" [P2], and
- full program requirements are not known from the start of a project and therefore type signatures will change a lot — this creates overhead when refactoring [P5].

Many participants mentioned wanting improvements to a strong type system's rigidity and "more flexibility when things are not quite right" [P69]. Current advanced type checkers tend to refuse to compile in order to stay 100% safe, which can cause frustrating errors. Participants would also like to see "more automation for dealing with 'human trivial' tasks" [P8], such as boilerplate code [P21] or "'trivial' [type] conversions like equalities on natural numbers" [P29]. Other improvements would include a "hammer" tactic [P7], which would try to brute force proofs, as well as "smarter termination checkers" [P22], which can better determine whether more complex recursion patterns are terminating.

(2) Improving the Ecosystems. By far the most mentioned desired improvements were to ecosystems of existing TyDD languages. These ranged from error reporting to integrated development environment (IDE) support, and also included documentation and community building. The desired improvements frequently highlighted features that are already integral components of mainstream programming language ecosystems.

Currently, "there are too few tools facilitating type-driven development, unlike the repertoire that object-oriented programmers enjoy" [P32]. Improvements mentioned by participants can be divided into five categories:

Documentation: Participants would like documentation for TyDD tools to be *richer*, *more searchable*, and *better integrated with types.* Currently, "documentation can be subpar" and it is sometimes "very hard to see whether a library definition could be used for [a developer's] purpose simply because its type is so generic that it is impossible to figure out at a glance whether the type can be instantiated to something [they are] interested in" [P5]. Having "illustrative, actionable docs with lots of meaningful, concrete examples" [P80] and "more easy to access educational resources" [P56] would help mitigate these issues.

Additionally, even if documentation does exist for a certain tool, it can be hard to find the answers and features the developer needs. For example, in Lean, programmers have "no effective way to discover what tactics are available" [P45] and Agda is missing a tool which lets programmers "search for functions based on their types" [P1]. In addition to wanting these general improvements to a language ecosystem, participants were asking for "better navigability / surveyability of library APIs during program development" [P18].

Finally, participants suggest that there could be "better integration of rich types and documentation" [P24]. With the proper IDE features, complex types could provide information to the programmer at relevant moments (e.g., inferred types). Conversely, "attaching information to types (notations, namespaces) and having IDEs / tools take advantage of that more comprehensively" could improve features such as suggestions and autocompletion [P21].

Error Reporting: In many programming languages with advanced type systems, "error message are usually very bad" [P17]. They "require knowledge of the tooling implementation" [P1], "expose internal concepts that are not very beginner-friendly" [P40], and do not easily "differentiate between actual logic errors and equivalents to 'missing semicolon' problems" [P15]. Specific changes participants would like to see are "more tools that suggest automated fixes" [P40] and the possibility to trace (erroneous) type-level computations [P67].

Visualisation: To help comprehend program and codebase structure, participants mentioned several types of visualisations they would like to have. Examples include "something like UML diagrams for type-driven development" [P32], "automated control graph construction / presentation" [P62], and "tools (even widgets) which enable exploring math ideas" [P13].

IDE Support: A significant portion of the entries pointed to editor support and how IDEs should present context, refactor code, generate code, and generalise these features by using Language Server Protocol[7] (LSP). Additionally, participants would like to see "wider editor support for interactive modes" [P16] and "better IDE support for large-scale proof engineering (e.g., being able to navigate theorem dependencies [and] displaying complex proofs in concise form)" [P219].

Typed holes are commonly used to present context to the programmer by displaying all available variables and definitions, as well as their type signatures. However, not all information being displayed is always relevant [P52], and "type checkers often have to introduce their own variables, but fail to explain why or where they were introduced" [P1]. Additionally, programmers "have to [successfully type check the partial program] to get [holes] to show something useful" [P176] — it would be nicer if context was available even before that.

Refactoring tools were also mentioned as lacking in the responses, with code extraction and signature adjusting as the two main examples. For example, participants would like "a

[7] A protocol for communication between language servers and IDEs [35].

good lemma extraction mechanism which allows [them] to create a new lemma from a proof state" [P63] as well as one "to add a new argument to a lemma so that all call sites are adjusted automatically" [P63].

Though code generation was listed as a benefit of TyDD, there were still improvements mentioned for this functionality: auto-complete should be smarter [P4], filter out suggestions that are nonsense [P162], and have better variable naming and formatting [P266]. Participants saw "a lot of potential in program synthesis and LLMs to help generate implementations, or present multiple possible implementations to the user" [P52].

Importantly, participants also felt strongly about the use of LSP to support many of these features. A participant wrote that "language servers should be seen as essential for the success of a [programming language] targeting the industry nowadays" and that "they should be part of the 'official package' " [P39]. Language maintainers should "focus on LSP functionality first, so that [developers] can use it in [their] editor of choice" [P63]. It would be beneficial to have generic LSP support for facilitating type-based interactions, such that not every LSP client for languages with such features has to "reinvent the wheel." [P6]

Community: Finally, participants would like to see TyDD language communities "regroup and build communities outside of academia by providing accessible pathway to newcomers with preexisting background in engineering" [P236]. This includes

- using naming conventions that are not at odds with OOP conventions (e.g., type*classes*) [P25],
- maintaining tools over time, such that users can safely incorporate them into their workflows [P17],
- making sure tools compose well with others [P17],
- providing "native data-structures (like arrays)" [P35] as well as more proof libraries for verifying algorithms [P55],
- defining standardised project templates [P220], and
- having code formatters / linters available to keep code style consistent automatically [P63] and to improve comprehension [P4].

Interestingly, Rust [28] and its ecosystem were often used as an example of what all TyDD tools should follow.

(3) Integrating into the Real World. While using type-driven development can have a variety of benefits for software engineers, many TyDD tools are not ready to be used for writing real-world software. Participants want a language in which they can prove things but also one that is "suitable for production code" [P220]. Currently, there seem to be four main deficiencies:

Interacting with the Real-World: Our participants would like "the ability to write programs that might be run in a business context (as opposed to an academic context)" [P28]. Currently, this is difficult in most strongly-typed languages, since focus is more on proving correctness of components rather than on frameworks and libraries for communication with the outside world (e.g., user input or networks). Participants asked for "more examples of use in practical software engineering" [P220] as well as "more software development and less pure mathematics" [P73].

DevOps Tooling: Additionally, having "more DevOps-related tools for actual deployment" [P73] is necessary to produce real-world applications with TyDD tools. The most mentioned tools were package managers, "like Rust with `rustup` and `cargo`" [P37]. "Every new cool language should copy the tooling provided by Rust and Lean [and] have a GHCup or `rustup`" [P37].

Integration with Other Languages and Tools: Participants indicated wanting to see "work on the interoperability with existing successful tools" [P236]. They would like "easier interfacing with other languages" [P4] as well as "widespread integration with automation like SMT solvers" [P47]. Ideally, "some degree of type-safety [would] extend across [the] whole stack" [P220]. This integration could also go the other way, where existing tools can improve TyDD tools. An example would be "a better AI companion that can be plugged deeper into the language, allowing it to utilize the proof state better than currently" [P29].

Performance: Performance is an often-mentioned issue with current strongly-typed languages — both at compile-time and at run-time. With stronger type systems, a lot of computation and checks are moved into the compilation step, meaning that languages that use them "tend to compile a bit slower, which worsens interactive development" [P5]. Run-time performance also seems to be insufficient, with many tools using "too much CPU [and] RAM" [P53]. Participants requested parallel compilation [P9] as well as "control over run-time representation" [P22] to help mitigate these issues.

(4) Adding More Features. There was a significant number of responses that requested either existing TyDD features to be brought to other language ecosystems or new features to be added to existing TyDD languages.

Participants would like to see "more dependent types in practical, industry languages" [P26] since it "would be nice if more languages leverage invariant specification like that" [P48]. They would also like "more hole orientation in all languages" [P48] and a "wider adoption of case splitting[8] and inhabitation search[9]" [P24].

Examples of more powerful features to add to existing TyDD ecosystems included "more interactive commands, e.g., for introducing a let-binding" [P0], "type-driven meta-programming" [P222], and "support for reflection on embedded languages" [P35]. Mathematicians asked for tools which "make translating math ideas to formal proofs less painful" [P13], such as supporting "shape polymorphism" [P35] or being able to "work with the internal language of various toposes" [P45].

5 DISCUSSION

With this work, we aimed to understand *how current practitioners experience the use type-driven development.* So, what does the practising community understand under the term?

RQ1.1: What does the practising community understand under the term "type-driven development"?

Based on the data, for current practitioners, type-driven development encompasses the usage of types in order to

[8]Case splitting is when the tooling can automatically split a hole into multiple new holes based on some data structure. For example, splitting on a list would result in one case for the empty list and one case for an element prepended to another list.

[9]A compiler can perform an "inhabitation search" in order to attempt to generate a potential implementation for a typed hole.

(1) *comprehend* the problem domain,
(2) *model* solutions to program requirements,
(3) *guide* developers to the implementation,
(4) *communicate* a program's intent and behaviour,
(5) *guarantee* a program's correctness, and
(6) *enrich* the context for developer tools.

These points correspond with Brady's [6, p. 5] definition of TyDD as being an approach where you "write types first and use those types to guide the definition of functions". He also writes that "you can think of types as being a *plan*, with the type checker acting as your guide, leading you to a working, robust program". Our community-based definition combines this with the concept of types as tools for comprehension and provides an overview of concrete benefits that practitioners perceive to be gaining when using TyDD.

RQ1.2: Which perceived benefits do practitioners gain from using type-driven development?

Firstly, our participants seem to use TyDD because it helps them with their mental models of the problem and their solution. The type annotations provide a basis for communication and comprehension and TyDD-derived tools such as context presentation or code generation allow them to hold only smaller parts of the program in their mind. This is also likely what contributes to the perception that type-annotated code is more maintainable. We can see a parallel here with other software engineering tools and paradigms:

- *domain-driven design* [15] focuses on a deep understanding and modelling of the business domain,
- *model-driven development* [21] allows developers to reason at a higher level of abstraction, letting them think about the domain rather than the implementation,
- *test-driven development* [44] encourages the developer to define the desired behaviour first, and
- *design by contract* can "help express the purpose of a [function] without reference to its implementation" [34, p. 46].

Secondly, the static guarantees provided by TyDD give practitioners a "peace of mind" and it seems that they feel more in control of the whole process. TyDD facilitates safer collaboration via clear contracts and encoded validation, and maintenance and refactoring are perceived as more controllable and less likely to introduce bugs.

Lastly, type-driven development also seems to make programming more enjoyable for these practitioners.

Since many of these benefits are more prominent in advanced type systems, we wanted to understand *what inhibits their adoption by a wider range of developers.* A logical place to start looking for an answer are the existing languages and tools facilitating TyDD, and determining how practitioners would like to see them improved.

RQ2: Which improvements would practitioners like to see in existing tools facilitating type-driven development?

From the identified improvements, we hypothesise that three form the main inhibitors to the adoption of advanced TyDD tools:

(1) the increasing effort required from developers to satisfy the type checker when using advanced type systems to express more complex constraints on their programs,
(2) the inadequate quality of current ecosystems for TyDD-oriented languages, and
(3) the insufficiency of libraries, tools, and performance required for building software which can interact with the real world.

We see a tension in the current TyDD tools between *usefulness* and *usability* (i.e., *ease of use*), the perception of which form the two key predictors of technology acceptance in the Technology Acceptance Model [55]. *Usefulness* determines whether a tool is valuable to its user, while *usability* refers to how easy it is for a user to use a certain tool. Though these two terms may seem similar, they are both important:

If a potentially *useful* feature requires such specialised knowledge or breaks so often that it is basically *unusable*, it does not actually provide any value to the user.
On the other hand, if an easily *usable* IDE feature is not applicable in any *useful* context, it might as well not be available.

Type-driven development and the associated tools are currently perceived as *useful*: they help developers think, and they can offer some strong guarantees. In many cases, though, they are not *usable* enough to be used by everyday developers. In Section 7 we discuss gradual types — an existing technical solution for reducing the rigidity associated with strict type systems. The work on gradual types for advanced type systems usually addresses its *usefulness* by demonstrating a proof of concept and making an argument as to what value it could bring. In most cases, though, it forgets about the *usability* aspect and does not integrate it into existing ecosystems nor verify the design with a user study.

6 LIMITATIONS & THREATS TO VALIDITY

Limitations in Study Design. While the survey setup let participants list benefits of and desired improvements to existing TyDD tools, it did not let them communicate the significance nor severity of their entries. Future studies could shed light on how impactful each benefit is and how to prioritise the suggested improvements.

Researcher Bias. Due to our methodology, we have certainly applied our own experience with type-driven development to the interpretation of the dataset. We acknowledge the interpretive nature of qualitative research and declare our constructivist epistemological stance, utilising guidelines on conducting socio-technical qualitative research to ensure the soundness of our work.

To reduce the likelihood of a misinterpretation that would pose a threat to the validity of our results, we used labels and diagrams during regular meetings [22, p. 260] to revisit each response multiple times in different contexts and to help us identify and understand the relationships between them. To ensure the credibility and rigour of our study, we follow Hoda's recommendations in Section 3 to explain how the participants were recruited, show how the data collection and analysis occurred, and share coding examples in this work [22, p. 338–339]. Additionally, we provide our complete dataset and codebook in a replication package [24]. Finally, as prescribed by the *Empirical Standards for Software Engineering Research*, we try to "present a clear chain of evidence from [participant] quotations to proposed concepts" [1] in Section 4.

External Validity. We have used convenience sampling to recruit our participants. Considering the channels that we have used for recruitment, our set of participants might be more positively inclined towards TyDD. As a result, our observations might not generalise to a wider population. Future work could replicate the study using stratified or random sampling to further validate the findings.

7 RELATED WORK

Research on how type systems are currently used and on their usability is currently still somewhat limited. Most of the work presented in the following section is from recent years, focusing on the directions we should look towards rather than on potential solutions to current problems.

Usage of Type Systems. Lubin and Chasins [30] utilised Grounded Theory to examine how programmers write code when using a statically-typed language. They conducted think-aloud programming sessions and semi-structured interviews, and their focus was on languages from the Haskell family. Like us, they found that programmers used an iterative process to model their domain and encode their solutions in type signatures, later using them to plan and decompose their task as well as to get feedback from the compiler. They also identified an "exploratory" style of programming when their participants were faced with a difficult or unknown domain.

Shi et al. [47] conducted an observation study on Rocq (then still named Coq) and Lean, which are both dependently-typed languages acting as interactive theorem provers. Their emphasis was on proof-writing specifically, focusing on the more advanced powers of the type system. Despite that, their findings were very similar to ours: participants interacted with the compiler and advanced by incorporating its feedback, they consulted resources other than the current proof, they considered design aspects beyond getting to a correct proof, and they struggled with "minutiae" when trying to convince the compiler that they are satisfying the constraints laid down by the types.

Usability of Type Systems. The two most relevant works to this paper are our cross-sectional study on the obstacles that new users face when learning to use Agda [25], and Gamboa et al.'s [19] study on the current barriers developers face in adopting and using LiquidHaskell [52]. Though focusing only on one type system each and emphasising entry barriers, the findings in these works correlate with the ones presented in this paper. Both Agda and LiquidHaskell users struggle with the underlying complexities of the system, and are not sufficiently shielded from them by the tooling. Inadequate ecosystems with difficult installation, unhelpful error messages, lacking documentation, and insufficient IDE support were a significant concern among both sets of participants.

Additionally, Crichton et al. [10] conducted a study on the misconceptions that users have about the ownership types in Rust. They found that their participants could, on a surface level, reason about why a program is ill-typed, but did not possess a deeper understanding of the resulting undefined behaviour. This seemed to be a key barrier in learning to use ownership types, which in turn prevented potential users from learning Rust — implying that users should be better shielded from such underlying complexities or have access to better tooling to ease working with them.

Finally, we would also like to mention the usability-oriented design for liquid types in Java by Gamboa et al. [18], which was created by conducting a developer survey where participants were asked to evaluate alternative designs for the syntax. In their work, they find that users must be able to build their program's specification incrementally, in a familiar, Java-like syntax, and that the design should reduce any unnecessary overhead for the programmer.

Gradual Typing. Gradual type systems [48, 49] aim to reduce the rigidity traditionally associated with static type systems by safely integrating static and dynamic typing into a single language. In a gradually-typed system, not all parts of the code have to immediately satisfy the type checker in order to be executed. This allows developers to

(1) already run tests and experiment with the code before putting in the effort of making it type-correct (though this is mainly useful when not applying TyDD);
(2) gradually add static types onto existing untyped code; and
(3) safely call untyped code from statically-typed programs.

The gradual type system guarantees that if a runtime type error does occur, it can be traced back to a specific location in the dynamically-typed program [56]. Examples of implementations of gradual typing for simple type systems include TypeScript [5], Hazel [40], and Typed Racket [43].

Gradual types are also being explored for more advanced type systems as a tool for providing a gradual path from simple to more expressive types. For example, being able to experiment with code in a dependently-typed language is even more important, since it is cheaper to discover simple bugs via testing rather than trying to prove correctness. Though there has been theoretical work on gradual versions of dependent types [14, 31], ownership types [46], security types [16], effect systems [3, 51], refinement types [29], liquid types [53], and session types [23], these systems have not yet been widely used in practice.

8 CONCLUSION & FUTURE WORK

Our first goal was to understand *how current practitioners experience the use type-driven development.* In our survey, we see practitioners defining it as an approach to program-writing in which they write type signatures first in order to gain a variety of benefits throughout their development process. Though the main selling point of having a type system is its ability to prevent a certain class of errors, our survey showed that current practitioners think TyDD can simplify the design and implementation process, improve communication and collaboration between developers, and create a more enjoyable and confident developer experience.

Our second goal was to determine *what inhibits its adoption by a wider range of developers.* It seems that though TyDD is viewed as a very *useful* approach, its tools are currently not considered *usable* enough. The more expressive a type system becomes, the more demanding its type checker is with its expectations from the developer. Additionally, because many of the advanced TyDD tools live in niche research languages, their ecosystems are not up to industry standards, and practitioners miss a lot of basic IDE support. Lastly, current TyDD tools do not have enough support for building applications which interact with the real world and their performance is not yet perceived as suitable for industry.

These findings are novel mainly because they now provide empirical evidence for what has thus far only been known from anecdotes. We have made the concept of type-driven development more concrete by providing a community-based definition and created an explicit list of both its benefits and the current inhibitors to its widespread use. All of these can now be used for communication between the programming languages and software engineering research communities, which could, through collaboration, make advanced TyDD tools available to a wider range of developers.

To achieve such a goal, we suggest a research agenda that encompasses a gradual increase in type expressivity within mainstream programming languages while still emphasising usability. Developers could then control just how much they wish to encode and consequently prove on the type level, and still receive sufficient support from their development environment. We define four actionable follow-ups for researchers and programming language designers and maintainers:

(1) *Bring user-oriented design to the forefront of type system research.* Building a strong, usable basis for the tools we already have would make them applicable in more contexts for a wider range of developers.
(2) *Investigate whether existing statically-typed languages could benefit from TyDD tools without having to change their type systems.* Hazel [40] is a good example of a language with a basic type system which leans heavily into hole-driven development, creating a live programming environment for its developers.
(3) *Incorporate type refinement directly into existing language compilers.* Refinement types are currently available in some mainstream programming languages as separate add-ons [9, 18, 45, 52, 54], but we hypothesise that incorporating them natively would allow for a more natural adoption.
(4) *Combine compile-time and runtime type checking.* Adding fully capable, complex types to an existing language without sacrificing its flexibility might be too ambitious a goal to reach directly, but combining them with generated runtime checks as is done in hybrid type checking [17] and gradual types might provide a good starting point.

ACKNOWLEDGEMENTS

We thank Daniel A. A. Pelsmaeker and Carolin Brandt for their reviews of this paper, which contributed to its overall quality of content as well as language. Additionally, we thank Ruben Backx and Wouter Swierstra, with whom we were collaborating on a parallel project. Our discussions about interactive theorem prover usage provided valuable insights and surely made their way into the ideas presented in this paper.

This research is partially sponsored by the Dutch Science Foundation NWO through the Vici "TestShift" project (`VI.C.182.032`).

REFERENCES

[1] 2020. Empirical Standards for Software Engineering Research. https://arxiv.org/abs/2010.03525v2
[2] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Manning.
[3] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A theory of gradual effect systems. *SIGPLAN Notices* (2014), 283–295. https://doi.org/10.1145/2692915.2628149
[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: which problems do they fix?. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014).* ACM, 202–211. https://doi.org/10.1145/2597073.2597082
[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer, Berlin, Heidelberg, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11
[6] Edwin Brady. 2017. *Type-Driven Development with Idris.* Manning.
[7] Edwin Charles Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Computer Science* 18, 3 (2017). https://doi.org/10.7494/csci.2017.18.3.1413
[8] Kathy Charmaz. 2014. *Constructing Grounded Theory* (2 ed.). Sage Publications.
[9] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA).* ACM, 587–606. https://doi.org/10.1145/2384616.2384659
[10] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Artifact for "A Grounded Conceptual Model for Ownership Types in Rust"* 7, OOPSLA2 (Oct. 2023), 265:1224–265:1252. https://doi.org/10.1145/3622841
[11] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. https://doi.org/10.1145/1995376.1995394
[12] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction (CADE-25)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26
[13] Edsger W. Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 10 (1972), 859–866. https://doi.org/10.1145/355604.361591
[14] Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate normalization for gradual dependent types. *ACM on Programming Languages* ICFP (2019), 88:1–88:30. https://doi.org/10.1145/3341692
[15] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.
[16] Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *2013 IEEE 26th Computer Security Foundations Symposium.* IEEE, 224–239. https://doi.org/10.1109/CSF.2013.22 ISSN: 2377-5459.
[17] Cormac Flanagan. 2006. Hybrid type checking. In *Symposium on Principles of programming languages (POPL '06).* ACM, 245–256. https://doi.org/10.1145/1111037.1111059
[18] Catarina Gamboa, Paulo Canelas, Christopher Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *International Conference on Software Engineering (ICSE).* IEEE, 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132
[19] 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
[20] Barney G. Glaser. 1978. *Theoretical sensitivity: Advances in the methodology of grounded theory.* Sociology Press.
[21] B. Hailpern and P. Tarr. 2006. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal* 45, 3 (2006), 451–461. https://doi.org/10.1147/sj.453.0451
[22] 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
[23] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. 2019. Gradual session types. *Journal of Functional Programming* (2019). https://doi.org/10.1017/S0956796819000169
[24] Sára Juhošová and Jesper Cockx. 2025. Type-Driven Development in Practice. https://doi.org/10.4121/8eda8cff-ac6b-47c8-908e-a99a81e08024
[25] 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
[26] Alexis King. 2019. Parse, don't validate. https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/
[27] Steve Klabnik and Carol Nichols. 2021. Understanding Ownership. In *The Rust Programming Language* (second ed.). No Starch Press, San Francisco, CA, USA.
[28] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language* (second ed.). No Starch Press, San Francisco, CA, USA.
[29] Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Symposium on Principles of Programming Languages (POPL '17).* ACM, 775–788. https://doi.org/10.1145/3009837.3009856

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

[31] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *ACM on Programming Languages* 6, ICFP (2022), 124:931–124:959. https://doi.org/10.1145/3547655

[32] Simon Marlow. 2010. *Haskell 2010: Language Report.* Technical Report. https://www.haskell.org/onlinereport/haskell2010/

[33] 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

[34] Bertrand Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[35] Microsoft. 2024. Language Server Protocol Specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

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

[37] Neil Mitchell. 2004. Hoogle. https://wiki.haskell.org/Hoogle

[38] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design: Recent Insights and Advances*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.). Springer, Berlin, Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7_6

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

[40] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languagess (SNAPL) (LIPIcs, Vol. 71).* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11:1–11:12.

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

[42] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[43] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. 2025. The Typed Racket Guide. https://docs.racket-lang.org/ts-guide/

[44] Adrian Santos, Sira Vegas, Oscar Dieste, Fernando Uyaguari, Ayşe Tosun, Davide Fucci, Burak Turhan, Giuseppe Scanniello, Simone Romano, Itir Karac, Marco Kuhrmann, Vladimir Mandić, Robert Ramač, Dietmar Pfahl, Christian Engblom, Jarno Kyykka, Kerli Rungi, Carolina Palomeque, Jaroslav Spisak, Markku Oivo, and Natalia Juristo. 2021. A family of experiments on test-driven development. *Empirical Software Engineering* 26, 3 (March 2021), 42. https://doi.org/10.1007/s10664-020-09895-8

[45] Georg Stefan Schmid and Viktor Kuncak. 2016. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the ACM SIGPLAN Symposium on Scala (SCALA).* ACM, 31–40. https://doi.org/10.1145/2998392.2998398

[46] Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

[47] 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

[48] Jeremy G Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop.* http://scheme2006.cs.uchicago.edu/13-siek.pdf

[49] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *Summit on Advances in Programming Languages (SNAPL)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset (Eds.), Vol. 32. Schloss Dagstuhl, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274 ISSN: 1868-8969.

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

[51] Matías Toro and Éric Tanter. 2015. Customizable gradual polymorphic effects for Scala. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, 935–953. https://doi.org/10.1145/2814270.2814315

[52] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem proving for all: equational reasoning in liquid Haskell (functional pearl). In *International Symposium on Haskell.* ACM, 132–144. https://doi.org/10.1145/3242744.3242756

[53] Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual liquid type inference. *ACM on Programming Languages* 2, OOPSLA (2018), 132:1–132:25. https://doi.org/10.1145/3276502

[54] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Programming Language Design and Implementation (PLDI).* ACM, 310–325. https://doi.org/10.1145/2908080.2908110

[55] Viswanath Venkatesh and Hillol Bala. 2008. Technology Acceptance Model 3 and a Research Agenda on Interventions. *Decision Sciences* 39, 2 (2008), 273–315. https://doi.org/10.1111/j.1540-5915.2008.00192.x Publisher: John Wiley & Sons, Ltd.

[56] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1