How Statically-Typed Functional Programmers Write Code

JUSTIN LUBIN, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

How working statically-typed functional programmers write code is largely understudied. And yet, a better understanding of developer practices could pave the way for the design of more useful and usable tooling, more ergonomic languages, and more effective on-ramps into programming communities. The goal of this work is to address this knowledge gap: to better understand the high-level authoring patterns that statically-typed functional programmers employ. We conducted a grounded theory analysis of 30 programming sessions of practicing statically-typed functional programmers, 15 of which also included a semi-structured interview. The theory we developed gives insight into how the specific affordances of statically-typed functional programming affect domain modeling, type construction, focusing techniques, exploratory and reasoning strategies, and expressions of intent. We conducted a set of quantitative lab experiments to validate our findings, including that statically-typed functional programmers often iterate between editing types and expressions, that they often run their compiler on code even when they know it will not successfully compile, and that they make textual program edits that reliably signal future edits that they intend to make. Lastly, we outline the implications of our findings for language and tool design. The success of this approach in revealing program authorship patterns suggests that the same methodology could be used to study other understudied programmer populations.

CCS Concepts: • **Human-centered computing → HCI theory, concepts and models**; • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: static types, functional programming, grounded theory, need-finding, interviews, qualitative, quantitative, mixed methods, randomized controlled trial

ACM Reference Format:
Justin Lubin and Sarah E. Chasins. 2021. How Statically-Typed Functional Programmers Write Code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 155 (October 2021), 30 pages. https://doi.org/10.1145/3485532

1 INTRODUCTION

Statically-typed functional programming languages like Haskell, OCaml, Elm, F#, and SML offer features and norms—like expressive type systems, strong static guarantees, and an emphasis on small and reusable functions free of side effects—that differentiate them from other classes of languages. These attributes are different enough from those found in more mainstream languages that they engender distinct modes of interaction between statically-typed functional programmers and their language, environment, and tooling. The aim of this work is to understand how the specific affordances of statically-typed functional programming affect the way programmers author code. The end goal is to deepen our understanding of an understudied programmer population, lay the foundation for evidence-based design of useful and usable languages and tools, and elucidate tacit community knowledge, which could ease the onboarding of new members to the community.

Authors' addresses: Justin Lubin, justinlubin@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Sarah E. Chasins, schasins@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA.

2475-1421/2021/10-ART155
https://doi.org/10.1145/3485532

We conducted a mixed-method investigation, beginning with a grounded theory analysis of 30 statically-typed functional programmers engaged in coding sessions, 15 of which occurred live over Zoom and included a semi-structured interview afterward. The other 15 sessions were drawn from livestreaming websites for increased ecological validity. Our theory (Section 4) unifies our observations in a framework consisting of four primary categories: (1) type construction, (2) focusing techniques, (3) hierarchical and opportunistic programming, and (4) reasoning and intent. After constructing this theory, we conducted a set of quantitative lab experiments (Section 5) to test some of our theory's predictions. Lastly, based on our theory and the results of our experiments, we outline some implications for tool design and unresolved questions in this area (Section 6).

This paper does not introduce or evaluate any new tools. Our theory development method was open-ended, driven by the goal of understanding the programming practices of working statically-typed functional programmers. The insights generated by our theory can be used not only as a foundation for improved languages and tooling, but also as a starting point for further interdisciplinary investigation into *why* statically-typed functional programmers write code the way that they do, whether or not their strategies could be improved, and whether the behavior of working functional programmers should shape how we teach beginners. Moreover, the methodology we used to develop this theory is generalizable to other understudied classes of programmers.

Contributions. In summary, we observed statically-typed functional programmers write code, interviewed them about their code authoring processes, and performed controlled experiments to test hypotheses generated by these methods. These endeavors resulted in four key contributions:

- A grounded theory of how statically-typed functional programmers author code, including their domain modeling, type construction, focusing techniques, exploratory and reasoning strategies, and expressions of intent.
- An experimental validation of hypotheses generated by this theory.
- A set of implications for tooling for statically-typed functional programming languages.
- A template for future need-finding studies of understudied programmer populations.

2 BACKGROUND

We first provide a brief background on two concepts this paper relies on.

2.1 Grounded Theory

Grounded theory methodology (GTM) is an approach to theory development with a long history in the social sciences that emphasizes an iterative cycle of data collection, data interpretation, and theory development (Glaser and Strauss, 1967). The goal of GTM is not to test a fixed hypothesis or existing theories, but rather to build up a *new* theory, grounded in data, to understand a phenomenon or domain. At a high level, it operates as follows:

1. The researcher collects data.
2. As the researcher collects data, they tag it with increasingly abstract categories that are, ideally, the simplest possible explanation for the data at hand.
3. As more data accumulates, this simplest possible explanation will grow more nuanced when unexpected or surprising new data challenges the developing theory.
4. This process finishes when the theory reliably accounts for the data being collected and no more "surprises" are found. (This is often called *theoretical saturation*.)

The key principle is *constant comparison* of data-with-data and data-with-theory (Charmaz, 2006, Glaser and Strauss, 1967, Urquhart and Fernandez, 2006), which allows the iteratively-developed theory to guide the data collection process by choosing data samples that will test the theory at its weakest points (Awbrey and Awbrey, 1995). Theories that survive this process of weakness-driven

data sampling are likely to be broad, robust, and predictive (Muller, 2014). Stol et al. (2016) give an overview of the use of GTM specifically in the context of software engineering research.

An example of the grounded theory process. To make GTM more concrete, we present here an example of how this process unfolded for developing the "Coding as a process of clarification" subsection in Section 4.3.2. All steps come from the standard grounded theory procedure and were interleaved with data collection.

1. First, when reviewing our recorded participant footage, we tagged hundreds of low-level descriptions of the activities occurring without any regard to abstraction. For example, two such tags were "pipeline forces top-down / generic thinking" and "rewrites to pipeline later if obvious pattern emerges."
2. In a second pass over all tagged materials, we grouped these descriptions into about 100 summary tags that group similar activities. The above two tags were grouped under "switches from explicit case to pipeline."
3. We abstracted these clusters into higher-level summaries, of which there were about 20. The above tag was grouped under "reuses prototype code." (This tag was later refactored into the idea of "coding as a process of clarification.")
4. Finally, these tags were grouped into the highest-level categories—those that appear as section headers in this paper. The above tag was grouped under "two modes and their harmony" (where these "modes" referred to hierarchical and opportunistic programming). This tag was eventually refactored to "interplay of hierarchical and opportunistic programming."
5. As a final step, after we wrote up the grounded theory, we reviewed our theory to derive hypotheses to test in future work (and in our quantitative evaluation in Section 5).

2.2 Hierarchical and Opportunistic Programming

An important distinction in the psychology of programming is between *hierarchical* and *opportunistic* patterns of program construction. Guindon (1990) and Davies (1991) characterize hierarchical programming as a systematic approach of refining the abstract to the concrete via a plan constructed ahead of time and opportunistic programming by deviations from this pattern (caused by, for example, a lack of conceptual clarity or tasks that exceed the capabilities of working memory). We discuss this distinction further in Section 8.2.1.

3 METHOD

Participants and recruitment. We conducted live study sessions with 15 participants with a variety of experience backgrounds, focusing mostly on those with at least two years of statically-typed functional programming experience. See Table 1 for details of each participant's background, years of experience with functional programming, and the language they used for the study. We screened participants (recruited from personal contacts, Slack workspaces, and Twitter) via a survey for (1) sufficient prior experience with statically-typed functional programming and (2) a diverse set of programming languages used.

Study protocol. Study sessions consisted of two back-to-back recorded sections of approximately 45 minutes each on a Zoom call with the first author. In the first half, participants narrated their thought processes aloud as they completed a subset of the tasks in Table 3 in a statically-typed functional programming language of their choice. We selected these tasks to be mostly open-ended and adapted them as time went on as a form of theoretical sampling to investigate particular themes that emerged. Additionally, as participants completed tasks, the first author incrementally asked them to complete follow-up tasks to investigate how they adapted their code. In the second half,

Table 1. **Participants we observed write code while thinking aloud and interviewed.** Exp. is experience with statically-typed functional programming. OSS is open source software.

ID	Language	Exp. (Years)	Exp. (Kind)
P1	SML	3	Academia
P2	OCaml	9	Academia, industry
P3	OCaml	3	Academia, industry, OSS
P4	OCaml	0.5	Academia
P5	Elm	3	Industry, OSS
P6	Elm	1	Industry
P7	Reason	2.5	Academia
P8	Elm	7	Industry
P9	Haskell	7	Academia
P10	Haskell	10	Industry, OSS
P11	Haskell	7	Academia
P12	F#	10	Industry, hobby
P13	PureScript	3	Academia, OSS
P14	BuckleScript	20	Industry, OSS
P15	Haskell	25	Academia

Table 2. **Livestreamers we observed write code.** Additional details available at https://github.com/plait-lab/stfp-livestreams.

ID	Language	Task
L1	Haskell	HTTP requests to GitHub API
L2	Elm	Board game with GUI
L3	Haskell	REST API for cryptography
L4	OCaml	Back-end web server
L5	Haskell	Advent of Code (coding challenge)
L6	PureScript	Front-end web development
L7	F#	Twitch chat bot
L8	PureScript	Networking for in-memory database
L9	OCaml	Ray tracer
L10	F#	Realtime number clicking game
L11	Haskell	PureScript language server
L12	Reason	Front-end for penetration testing tool
L13	Elm	Narrative engine for interactive stories
L14	F#	Full-stack web development
L15	F#	City management game

Table 3. **Tasks for grounded theory sessions.** These tasks were open-ended by design, and we often made slight per-participant modifications in accordance with the constant comparison and theoretical sampling aspects of GTM. We presented each numbered sub-task only after participants completed the previous one.

Description	Purpose
Calculator. 1) Implement a calculator that takes in an expression representing a mathematical computation and evaluates it. The calculator should support addition, subtraction, and multiplication of integers. 2) Extend the addition operation so that multiple integers can be added at once. 3) Extend the calculator to support division. 4) Extend the calculator to include an operation that returns the last calculation's answer.	To observe how participants refactor their code to handle evolving requirements, especially in the presence of effects such as partiality (division) and mutable state (calculator history).
Geometry API. 1) Implement a function that, given two circles, determines if they intersect. 2) Implement a function that, given two lines, determines if they are parallel. 3) Implement a function that, given a shape, prints out a description of what kind of shape it is.	To observe how participants reconcile different type modeling strategies. (Each shape was initially presented as independent, but later both must be handled by the description function.)
String manipulation. Create a function that takes a number and returns a list of strings containing the number cut off at each digit. For example, when called with `1080`, the function should return `["1", "10", "108", "1080"]`. (Drawn from Barke et al. (2020).)	To observe how participants decompose tasks when dealing with relatively unstructured data types (integers and strings), especially paying attention to their use of pipelines.
Tic-tac-toe. Implement a 3×3 tic-tac-toe game.	To observe how participants settle on type representations, how they decompose larger tasks, and how they maintain invariants throughout their programs (such as turn order, win conditions, and validity of moves).

the first author asked participants in a semi-structured interview to talk about their experience with statically-typed functional programming and to elaborate on particular topics that came up during their narration.

Livestreamed programmers. Operating on the premise that livestreamers' programming tasks are more diverse and more representative of actual tasks faced by statically-typed functional programmers, we also analyzed video clips approximately an hour in length from 15 statically-typed functional programmers who publicly livestreamed coding sessions. See Table 2 for summaries of the programming task in each clip and the programming language used. The clips were selected manually at the first author's discretion specifically to screen for diverse languages and tasks. Livestreamers' task diversity also had the benefit of introducing an element of randomness into our theoretical sampling, which ultimately influenced the tasks for the live sessions.

Analysis. We used a constructivist grounded theory approach (Charmaz, 2006) to analyze the video recordings. The first author began the process of grounded theory analysis for each video by tagging "chunks" of the videos with specific summaries, thereafter grouping these chunks by similarity into open codes while memoing and discussing findings with the second author. As we continued data collection, we built up larger categories until identifying the key categories at play in the programmers' authoring processes. We ensured that we had reached theoretical saturation (as signaled by a lack of surprises or unexplored consequences) before ceasing data collection.

4 THEORY

Our grounded theory analysis revealed that:

(§4.1) Participants often started a task by iteratively constructing types to model their problem domain and encode design decisions.
(§4.2) Participants then leveraged types to help themselves focus by (i) relying on the compiler as an assistant and (ii) using types to plan and decompose their tasks.
(§4.3) Sometimes, when faced with a difficult or unknown problem domain, participants complemented this systematic style of programming with a more exploratory one, the details of which were highly varied in comparison to the systematic style.
(§4.4) Lastly, no matter the style of programming, participants reasoned about code differently and expressed their authorship intent in diverse ways, not all via valid code.

Based on these observations, we identify a set of hypotheses in outlined boxes throughout this section. The subset of these hypotheses that are also numbered and styled with a shaded background form the basis for the quantitative lab experiments in Section 5; we selected these hypotheses based on our subjective assessment of which would be most actionable for tool and language designers.

Some of these observations are not necessarily specific to statically-typed functional programmers, but this section describes how they play out specifically in the statically-typed functional programming process; the goal of this work is to understand and characterize the statically-typed functional programming community rather than to identify particular patterns as exclusive to them. We speculate that some of the insights described here will generalize to non-functional programming languages with rich static type systems such as Swift and Rust as well as dynamically-typed functional programming languages such as Racket, Clojure, and Erlang; studying these communities and performing a comparative analysis would be an exciting avenue for future research.

4.1 Type Construction

P2 stressed the importance of using types for domain modeling:

> That's usually my first instinct … let me understand my domain, let's write down what these things are, let's give them a name, let's organize them appropriately, *and then* we can start to define our behavior on top of these things.

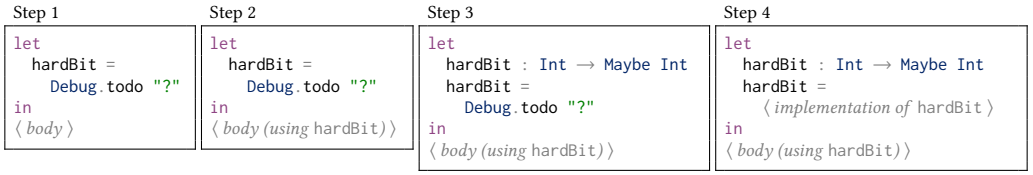

Fig. 1. **Constructing a "hard bit" of a function.** P8 (1) introduces the hard bit in a let binding, then (2) uses the new variable in the body. This usage helps them (3) deduce an explicit type for the variable, which helps them (4) implement its definition.

P6 noted that types help them detect design flaws early in the process, adding: "Once you get the types right, the rest of [the program] follows pretty quickly." Given the perceived benefits of type-driven development, one might wonder: *How are types constructed "right" in the first place?*

4.1.1 Iteratively Constructing Types and Expressions. For many participants, type construction was not just a linear (or even branching) process, but, rather, a cyclical one, often including the construction of expressions before the types had solidified. For example, participants wrote example expressions *before and during* the construction of their types (P4, P6, P8), basing their decisions of how to structure their types on how they structured their expressions. P7 took an iterative, usage-centric approach, noting that although they first think "From a simple/conceptual/ontological kind of level: How should these types be?" they sometimes realize "Oh, okay, wow this is actually way more painful. Like, it felt right to me from just like a lofty, abstract perspective, but now that I'm getting into the nitty-gritty … ergonomically, it's too costly." Similarly, P10 noted they often choose the simplest type representation that supports the operations they need for the task at hand (such as representing mutable state as a function); then, as they code and begin to understand the problem domain better, they switch the type to a more specific one (such as representing states specifically as a dictionary or finite-state machine). Figure 1 depicts another kind of iteration between types, signatures, and expressions: P8's handling of a "hard bit" of a function, which they pithily characterize as "[Having] the usage drive the type rather than the type drive the usage."

Hypothesis 1. When constructing new programs, statically-typed functional programmers iterate between editing types and editing expressions.

4.1.2 Feeling Odd When Types Are Amiss. At times, participants reported feeling "very wrong" (P4) or that something felt "weird" (P5) or "fishy" (P7) while coding. Ultimately, each time a participant reporting feeling this way, it could be traced to types they viewed as unsatisfactory: in all cases, participants either identified particular discomforting aspects of how they modeled their types or reported being more at ease after modifying their types. One prominent trigger for these feelings was *data representation redundancy*, which both P7 and P8 explicitly called out as particularly undesirable, recalling that this phenomenon had lead to bugs in the past. As a consequence of their feeling of personal discomfort, participants were able to diagnose and repair issues with their types.

Hypothesis. Statically-typed functional programmers experience personal discomfort when working with redundant type representations.

4.2 Focusing Techniques

> No, they're pairs! What am I doing? Yeah, ugh, okay—I'm trying to do too many things at once. That's also a pretty common thing. (P3)

Part of the challenge of programming is managing *mental workload*, or "the amount of mental work or effort necessary for a person … to complete a task over a given period of time," which "is

not merely a property of the task, but also of the individual, and their interaction" (Xie and Salvendy, 2000). Some of the variance in mental workload comes from individuals' ability to manage their *working memory*, a "temporary storage system under attentional control that underpins our capacity for complex thought" (Baddeley, 2007, Chapter 1). To complete complex tasks, programmers must focus their attention on relevant information, reducing extraneous mental burden.

Although a rigorous workload analysis would be required to determine whether the following strategies are actually effective, we observed (and participants described) a few focusing techniques that are made possible by the nature of statically-typed functional programming languages.

4.2.1 Relying on Compilers as Assistants. There were two main ways in which participants relied on their compilers as assistants:[1] as *corrective* tools and as *directive* tools.

Compilers as corrective tools. Participants often invoked their compilers on code that they believed to be correct, only to be met with a stream of error messages. While these errors could be viewed as obstacles to having the computer run their programs, participants commonly stressed that this corrective behavior of the compiler was useful not just to fix their programs, but also to correct their *mental models* of the problem domain. P11 described such a process:

> Every few functions I'll be like "Oh, have I correctly seen the relationship between all of the things I just wrote and the data types that I actually care about?" by loading it in and seeing if all the types match up.

P14 reinforced this idea, noting that feedback from the compiler was helpful first for correcting their mental model, then, later, for correcting their actual program:

> To me, the type errors are an indication that I don't understand what I'm trying to put in next correctly more than a hint or guardrails per se. And then they become [guardrails] when I say "Okay, this what I believe I want, now tell me how I did it wrong," and then I'll go and fix whatever it was that mismatched.

L4 demonstrated the extent to which compiler errors are integral to the development process, noting, in the absence of negative feedback from the compiler, "I'm a little troubled because … I expect the compiler to get mad at me right now, and it's not getting mad at me, and that makes me nervous." P1 echoed this sentiment, noting their discomfort at situations in which the compiler cannot give negative feedback: "Lifting an auxiliary function like that can be fraught, because sometimes the auxiliary function and the original function have the same type … and this is not necessarily something the type checker can fix for you." Participants felt similarly about the use of wildcards in pattern matching, which silently assign pre-existing behavior to new variants of an enumeration. P7 mentioned that, in their main codebase, wildcards are *completely disallowed* for this reason, even though they make programming more convenient.

Compilers as directive tools. For many participants, error messages from their compiler served as an auto-updating to-do list, directing their next steps in implementation. One instance of this pattern is *hole-driven development* or *sketching* (discussed further in Section 4.2.3), what P10 described as a "more advanced version" of interacting with the compiler:

> If I kinda know what I'm doing in a problem, … I [write] the whole thing using my mental model, and then I will run the compiler, and it will tell me "Oh, you forgot to put a `join` in there," and then I will fix those things … The more advanced version is, when I'm not sure what I'm doing, I will put holes in … I'll be continuously building the program with the compiler at each point telling me what I haven't yet solved.

Another common and related compiler-directed workflow took the following form: (1) the programmer expresses their intent to complete a program transformation by modifying a small part of the program and compiling, then (2) the compiler produces a list of sub-tasks (in the form of

[1] With thanks to Evan Czaplicki for popularizing the phrase "compilers as assistants": https://elm-lang.org/news/compilers-as-assistants.

```
-- Set (Position, Position)
-- Set (Position, Tile, Maybe Color, Position)
-- Board, Set (Maybe Color, Position)
-- Board
```

Fig. 2. **Following the types.** L2 "followed the types" by writing down the types for each step of moving a board game piece in comments (shown here), then implementing each one sequentially.

error messages) to complete the transformation—the programmer can simply *follow the errors*. P1 described this process for modifying a type:

> It broke a lot of stuff, but it was super nice because I made the change, and the type checker told me: "This no longer type checks; here's all the places where it goes wrong." And I can just go through and fix all the type errors, and that essentially is doing the change.

L13 used this strategy extensively while refactoring a type that relied on maintaining a list and an index into a type that relied on just a zipper.

A source of variation between participants was *in what order* they completed the sub-tasks set forth by "following the errors." Participants often first performed semantics-preserving refactors to get their code into a clean, working state before introducing new semantics—but that was not always the case. For example, when adding a possibly-failing division operation to their calculator, P8 simultaneously made their function return a `Result` *and* implemented new functionality before refactoring other parts of the function to handle the possible failures. They noted that they did so because they knew how the error handling would work for division, but not for the other cases. Similarly, P7 added new semantics to a function *first* as a lightweight way to make sure their code transformation would have the desired effect and ergonomics, thus avoiding committing to an arduous sequence of edits too early.

Hypothesis. When statically-typed functional programmers refactor a type definition, they use the compiler error messages as a to-do list.

Many participants went so far as to compile for the sole purpose of looking at error messages, *fully aware that the code they wrote was invalid*. For example, P3 narrated "I'm going to compile even though I know it's not going to work" and later mentioned "I'm expecting to get a type error here" as they compiled. Similarly, when faced with figuring out an unknown type, P14 mentioned that they would often "write a placeholder function of my own with a deliberately broken type and then shove that in there, knowing that that can't be misinterpreted as correct under any circumstances."

Furthermore, participants went out of their way to give information to the compiler that would result in better error messages. For example, when stuck on an error that dealt with a complex, nested type, P6 added a type signature to a function to get a more specific error message. Similarly, as P3 grappled with type errors relating to OCaml's `List.fold_left` library function, they narrated "This thing already knows it wants to be an `int`, so let me verify my assumptions here—this should be an `exp`, this should be an `int` …", adding extraneous type signatures to expressions as they spoke, later removing them once they fixed the errors. As a more complex example, P5 mentioned that they knew that the Elm compiler infers a type for a `case` expression based on its first branch, so they re-ordered code they knew was correctly-typed to be the first branch to get a better type error message on a more complicated branch.

Hypothesis 2. Statically-typed functional programmers run their compilers on code they know will not compile.

4.2.2 Type-Directed Top-Down Decomposition. Participants frequently used types to structure their top-down decomposition of problems, a strategy many participants referred to as *following the types*. As P9 put it:

> Usually I will write a type annotation for each function I am trying to implement before I start implementing the function, which is like the #1 step in implementation, which I definitely use all the time.

L2 exemplified this pattern more granularly by writing down the types of each step of a desired algorithm, then later filling in their implementations (Figure 2). Similarly, P8 knew they wanted to use the higher-order library function `List.indexedMap` but was unsure how to implement its function argument, so they copied and pasted the type signature from the documentation into their code and implemented the function top-down from the signature.

P7 stressed the importance of following the types in maintaining their focus:

> I find types really help me just organize … it's very important for me to be able to just chunk different parts of functionality, and … once I've gotten down the type signature, I can just focus on this function and not think about really anything outside of it.

P3 highlighted that working top-down from the type lets them think more clearly about sub-tasks: "Just like, *actually typing this out*, and sitting here, kinda freed my brain of this outer context."

One instance of this pattern was *leveraging parametricity*. While implementing the string manipulation task, P7 realized they needed to access the last element of a string (represented as a list of characters). Rather than folding this functionality into the recursion they were already writing, they implemented a parametric function that, given a list, returns its last element and the remainder of the list. Extracting a parametric core of this task let P7 focus on exactly *how* to shuffle around the list data without regard to *what* that data was, apparently reducing their mental workload.

These strategies provide insight into why statically-typed functional programmers place so much emphasis on having their types model their domain as precisely as possible. When types are "on the nose," P10 explains, programmers can often rely on types when they are stuck to essentially dictate exactly what expression to write out—a strategy that is augmented by the use of guiding feedback from the compiler in the presence of typed holes. As P1 noted, leveraging parametricity and typed holes proves to be a powerful combination:

> Often, for nice enough programs, parametricity gets you all the way there, by just, you know: there's only three things you can plug into the hole. So that's another tip that I take pretty seriously—is using typed holes and is using parametricity to my advantage very strongly.

4.2.3 Sketching. Program sketching (Solar-Lezama et al., 2005) is a practice in which programmers leave parts of their code unimplemented; typically, low-level details are omitted for the purpose of focusing on higher-level intent. Program sketches are frequently used as specifications for program synthesis (Gissurarson, 2018, Lubin et al., 2020, Solar-Lezama et al., 2006, Torlak and Bodik, 2014), but program sketching even without any automated feedback was a valuable pattern for participants. In tandem with type-directed, top-down decomposition, participants often used some notion of a well-typed "program hole," even if it was not explicitly supported by the language. P1 emphasized that typed holes are "pretty much invaluable to [their] development process, to the point where the fact that OCaml doesn't have them makes it kinda painful to use sometimes."

Participants reported various benefits to sketching, including that it decomposes the problem into simpler sub-tasks, that it helps structure the overall solution, and that it serves as a reminder for what to do next, ensuring that the programmer stays focused and does not forget the tasks at hand. P5 extensively used the names of yet-to-be-defined functions and types as named holes while working on the task at hand; regarding this general strategy, P11 noted that it is "a good way to keep track of what is left to do to just have a list of functions that don't exist and go through and do them." Similarly, both P4 and P5 mentioned that some notion of an incomplete placeholder is useful

as a way to avoid forgetting other tasks while focusing on the one at hand, such as when working on a `case` expression with many branches. P10 took this idea to the extreme, implementing their tic-tac-toe game by first sketching out all the "driver" IO code (which contained many holes in the form of references to yet-to-be-defined functions), then going back and providing type signatures for each of these functions and implementing them; regarding this tactic, P10 remarked that "You might have forgotten that there's an entire part of your program that needs to be written, basically. And the only way to expose that fact is by writing, essentially, the top-level driver …"

But not all participants sketched out their top-level program *in code*; participants often reported constructing such a sketch *mentally*. For example, P15 implemented their tic-tac-toe game by first writing out all the type signatures for the "atoms" of their program, then implementing them, then combining them all in a top-level driver. While this behavior visually resembled opportunistic construction of programs (writing down everything possible in an ad hoc or piecemeal way, then gluing it all together at the end, as in Section 4.3), P15's reported experience was exactly the opposite: the driver was simple enough to be completely sketched out in their head, enabling them to start writing signatures for the atoms immediately, rather than actually writing out the sketch:

> When it's very clear what the high-level loop is or [what] the high-level decomposition is going to be, then I can start working with the atoms. But if it wasn't so clear—and there are many situations where that's not clear—there, it's more like there's this big rock and I just try and hack off smaller bits, and I keep trying to hack off smaller bits until, okay, it looks like these are the different primitives that I'm gonna need.

P9 reported a similar approach: "When I program, I usually have this top-down approach, so I'll have a bigger function which will do some big thing, then I'll decompose it in my head and then implement the functions for the sub-tasks and then compose them together." These examples indicate that even having access to a granular, textual reconstruction of code as it was written would not be sufficient for distinguishing certain authoring patterns (such as decomposition by mental sketching vs. opportunistic conglomeration of atoms).

Hole filling order. As with fixing compiler errors, participants engaged in substantially different ways of *filling* these holes. Three general strategies that emerged were *breadth-first*, *depth-first*, and *hard-first*. In breadth-first hole filling, the programmer sketches out a skeleton of a solution of the task they are currently working on at a single level of abstraction, then later fills in the missing details. For example, when talking about writing part of the main IO loop for their tic-tac-toe game (a function `askPlayer`), P9 mentioned:

> The problem with the parsing is that you have to come up with the format in which the user have to write the input … this is a decision which I did not want to make right now, because it will take some time and I'll lose the focus of the `askPlayer` function.

P13 pointed out that another benefit of breadth-first hole filling is that it defers tedious implementation details until later in the development process, at which point the program types are more likely to have stabilized. Specifically, regarding the implementation of low-level details, they note: "I have to spend time doing that, and then if I want to change it later, it's like, wow, I just wasted so much time." Lastly, P10 noted that simply leaving parts of a program that they do not yet know how to implement unfilled for some time (as in breadth-first hole filling) can make the hole easier to fill for two reasons: first, that "there might be things that you discover or learn while you're implementing the easier code," and, second, that "there's also just the benefit of time … your brain is constantly thinking of things in the background, and if you spend ten minutes doing something else, then you might realize during that ten minutes that the approach you had been taking is wrong."

On the other hand, in depth-first hole filling, the programmer refers to a variable before defining it, but very soon thereafter goes to fill the hole, going down a level of abstraction before completing the current function. P14 notes that it is helpful to leave parts of the program unimplemented to start

but stresses the importance of getting a program up and running as soon as possible—something not typically possible with a partial sketch during breadth-first hole filling. Regarding filling holes, they noted: "If it's not distracting, then I'll probably write it right away to get it out of the way" so that they can get their program running sooner. P12 noted that depth-first hole filling better matched their cognitive process *outside* of programming, so it came more naturally to them:

> The main reason is I often think in this way on my own anyway. I get some idea and kinda digress and explore [that] idea … and then, kind of, go back to whatever was going on before in my head.

Lastly, P10 pointed out a hybrid between breadth- and depth-first hole filling: hard-first hole filling, in which the most difficult holes are filled first. They described this as their default strategy because it reduces their "schedule uncertainty," improving their predictions of how long tasks will take by surfacing the difficult parts and the sub-tasks they entail early in the process. In short, their justification for this strategy was: "Hard problems beget more problems."

Hypothesis. When decomposing a task, statically-typed functional programmers commonly construct a typed sketch either mentally or textually.

4.3 Hierarchical and Opportunistic Programming

Participants primarily engaged in hierarchical programming by "following the types" (Section 4.2.2). The opportunistic deviations from this pattern, however, were far more varied. These opportunistic strategies were not at odds with hierarchical ones, and, in fact often complemented the hierarchical ones harmoniously, with participants switching frequently and fluidly between the two.

4.3.1 Opportunistic Strategies. Participants employed varied opportunistic strategies, including:

Constructing expressions bottom-up. Participants sometimes constructed expressions by iteratively wrapping expressions with more complex structures. For example, L1 first implemented a function's core logic in a `do` block, then wrapped the entire expression in a function called `loop` when it called for recursion (as did P4 for their calculator). Similarly, after implementing a function's core logic, P5 wrapped it in a `case` expression that handled errors. Participants also used holes when iteratively constructing code bottom-up;for example, L5 iteratively constructed a function around an `undefined` value in Haskell with an explicit type signature, filling it in only after the rest of the task at hand was completed.

Another common way for participants to build expressions bottom-up was to iteratively construct a "pipeline" (i.e., composition) of functions, especially when the types were concrete. For example, L1 built up a string processing pipeline one function at a time, testing each iteration in the REPL, finally landing on (approximately) `map (strip ∘ head ∘ splitOn ";") $ splitOn "," url`.

Using the top level of their file as a notepad. Participants sometimes directly typed what was on their mind into the top level of their file without regard to syntax; the top level thus became a textual locus for brainstorming. Regarding this behavior, P13 commented:

> I don't really think of it as "top-level," it's more like, right now, top-level is the notepad for me, and it's not even part of the code. So, at the beginning, I'm not really thinking of this file as something that's even runnable. It's just like note-taking … like, I could just do this all in a `txt` [file], … then when I feel ready I'll transfer it onto here.

P10 noted the utility of this approach for constructing pipelines—"I will realize that I need a pipeline of things in a function … sometimes I literally just start writing out elements of the pipeline in a syntactically invalid way"—but also stressed that they try to minimize the amount of time their source file is syntactically invalid because their compiler cannot provide feedback during such times. P15 demarcated two distinct stages of their program authorship: an ideation stage, in which they

draft fragments of code at the top level (not treating the file as runnable), then an implementation phase, in which they actively seek compiler feedback to ensure their code is correct.

Other participants took different approaches to this same general strategy. L1 directly coded a rough draft of a function at the top level of the file, then used it as a reference for the real implementation later. P4 reasoned about their calculator function first as just a pattern match at the top level, then later tidied it into a formal function. L4 took a more lightweight approach, using a sketch at the top level only to query complicated type information from the compiler.

Performing pattern matching to handle sub-cases independently. Participants often introduced pattern matches that would be unnecessary with the use of more idiomatic combinators, but which let them visually examine and handle sub-cases independently. For example, P14 used a `match` expression to decompose tic-tac-toe winning conditions, adding and implementing each branch one-by-one as they thought of different cases until the `match` was exhaustive. P8 explained that an explicit `case` expression is helpful in these cases:

> I really prefer the pipeline for readability, but it's a little bit harder to write because I don't feel like I have the streamlined version of how it's going to look, whereas if I write the `case` down, it's telling me, here's all the possibilities, just handle them—like, start with this and go to the next one.

Hypothesis 3. Statically-typed functional programmers face reduced workload when implementing a task with explicit pattern matches as compared to with combinators.

4.3.2 Interplay of Hierarchical and Opportunistic Programming. Participants often used opportunistic strategies in concert with hierarchical strategies. P3 described how they do so, writing code that meets in the middle:

> I kind of understand, maybe, what I've got, so I can do some bottom-up exploration. And I pretty much know where I want to be—which is the type signatures—and it allows me to do some top-down programming. And when it's not clear to me how to connect the two, and … I'm not feeling super productive or I feel stuck trying to think from one end, I just switch to the other to try to glean some more context.

P11 reported a similar harmony between opportunistically writing down everything that came to mind and hierarchically ordering their program:

> For something … where I haven't thought a ton about it up front, I start by writing down a bunch of the sort of data that I know I need and the primitives on it … And then I'm switching over to the "Okay, well, I know what the data looks like, and I know how to work with the data," and now I'm doing the interaction, and so for the interaction I tend to think of it top-down as, like, "Okay, well, what does the overall structure of the interaction look like?"

Relying on both hierarchical and opportunistic strategies also helped participants overcome some of the indecision that came with deciding how to structure their types. For example, P7 remarked:

> Maybe I already spent a lot of time trying to think about how to do it the right way just in my head, and I'm not getting anywhere. Then, at that point, I'm just going to be like, okay, it's time to push through and suppress my perfectionist tendencies of stripping away all redundant things.

Coding as a process of clarification. The urge to "just push though" reflects a larger pattern: participants saw programming as a *process of clarification* rather than just a transfer of code from their mind to their editor. For example, P3 highlighted the importance of planning in their coding process, but also included pushing through with "exploratory" code as "planning." P8 mentioned they often start with many pattern matches (as in Section 4.3.1), then rewrite the code to be more elegant in a bottom-up fashion once it is functional and its patterns are more easily recognizable. They noted that once the `case` expressions are written, "You can start at the deepest level and sort of streamline them." P10's breadth-first sketching strategy (Section 4.2.3) exhibits this same

clarification pattern: they *hierarchically* decompose a problem with a sketch, then *opportunistically* work on the easier sub-tasks to gain information about how to solve the harder sub-tasks.

4.4 Reasoning and Intent

4.4.1 Diversity of Reasoning Approaches: What Feels Natural, Easy, or Right. Approaches to reasoning about a problem domain differed significantly from participant to participant and were often dependent on particular programming contexts. For example, P8 introduced local, argument-passing state to their calculator in Elm to implement history, but mentioned that they would model state completely differently in a web app (their primary use for Elm). For the same task, P5 did not even consider a mutable reference, whereas P1 wanted to use one immediately (but ultimately settled on explicit state passing). P4 initially conceptualized the calculator history as a specialized binding context, but decided to use a mutable reference for ease of implementation (which they felt to be unsatisfactory). P15 described how they try to recast tasks as instances of problems they can solve using building blocks ingrained in their programming process:

> I'm trying to take whatever my giant problem is and … I have a particular kind of funnel … and I'm trying to squeeze the problem that I have—mold it so that it fits into this particular kind of funnel.

This diversity hints at the trade-off between what feels (1) natural, (2) easy, or (3) right. For example, mutable references were easy yet felt wrong to P1 and P4 in the previous paragraph, but were natural to P1 and not to P4. As a final example, P3 was working in a context that required an integer, but they realized that the computation might fail; their initial reaction was simply to use -1 to signal failure, but they could not bring themself to do so. (It felt natural and easy, but not right.)

Hypothesis. Statically-typed functional programmers have different approaches to reasoning about problem domains (including what feels natural vs. easy vs. right) even for the same task and language. Moreover, the same programmer may have different reasoning approaches for a task in different contexts.

4.4.2 Reasoning About Code Essence. Compilers treat programs as formal objects, but participants often engaged in authoring patterns that demonstrated reasoning about their programs more intuitively. For example, P10 mentioned that it is important to them to have a "working program as soon as possible," but, in response to a question about what constitutes a "working program," P10 explained that sometimes programs that are not even runnable can be considered "working": "I would say that if there's a hole … and I know exactly how to implement it, then … from my mental point of view, it's working." More generally, participants employed programming patterns that did not rely on or result in valid or desirable code; nonetheless, they were helpful for participants to concretize and refine their intent about the *essence* of the code. Such patterns include:

Reasoning purely despite effects. P8 applied `List.sum` to a `Result` (rather than using `Result.map` first) and L8 used the result of a computation that might fail (encoded with `Maybe`) in a composition of operations without handling the `Nothing` case. L1 and L5 used `map` to apply a function to a list, but later realized that the iteration needed to stop early, rewriting the code to handle the "effect" of early termination. At the pattern level, L15 matched values of an `option` type with the patterns `None` and `Invalid _`, later correcting the second pattern to `Some (Invalid _)`. At the type level, L1 reasoned about an "infinite [self-]composition" of a function of type `Maybe ETag → IO (ETag, [Pull])`, then later had to translate this intuition to code.

Conflating a value with its image under a function. Multiple participants identified values in their problem domain with different representations of the same concept, leading to formally invalid but not implausible code. For example, when checking for a division by zero in their calculator, P3, P5, and P8 first checked if the right-hand side of the division was equal to the integer literal `0`,

even though it was actually an expression in their calculator grammar (i.e., forgetting to use `eval`). Similarly, L1 conflated an HTTP request with its string representation (i.e., forgetting to use `show`).

Checking only for patterns of interest. Participants often used pattern matching to handle just one case, using a wildcard to discard all others. For example, when working with pairs of computations that might fail, both P1 and P5 immediately introduced a pattern match expression and handled the "success" case first, only adding the error handling afterward by using a wildcard as a catch-all.

Although wildcards reduce the effort of handling error branches, they are in direct tension with participants' desire to use compilers as assistants. As mentioned in Section 4.2.1, wildcards prevent the compiler from statically checking that new variants are handled correctly—a fact that P1 noted:

> I have some code where I have a bunch of cases, and then I think I'm at the end … and then I wildcard the last pattern … then I add a new variant to the type, and then suddenly I compile with no errors, but then whenever I use that new constructor, it falls off the end of the function.

4.4.3 Signaling Intent. Participants often had high-level patterns in mind that they needed to translate into executable code. For example, P1 reflected on adding history to their calculator:

> I had a pretty good idea of what I was going to do. The main question was just: How do I represent the state? It seemed pretty clear that the pattern was going to be: thread some state around … [then] add a case to the evaluator where, when you hit this case, you dereference the cell and grab the data.

For this task, P2 noted that the type system made them "think, at each layer, who needs to have access to the state and who needs to update it." Similarly, L8 initially implemented a key-value store using argument-passing to model mutable state but later switched this representation to a reference cell (a long process that started by modifying the program's types).

Although these high-level plans entailed many low-level edits (e.g., introducing parameters and updating call sites), participants initially expressed their intent via modifications to types and signatures. These expressions of intent can be interpreted as *signals*, with the corresponding executions of these intents as *responses*. For example, using yet-to-be-defined functions when sketching (as in Section 4.2.3) can be viewed as a signal; the response would then be filling in the holes later. P5 had their editor leverage this signaling behavior: they would often reference undefined functions, and their editor would automatically offer to define them with a blank implementation.

Sometimes, though, responses to signals never came. For example, some participants noted that the type system was not always expressive enough to capture their intent, often when trying to "make illegal states unrepresentable."[2] In such cases, participants signaled their intent that certain states should be illegal by pattern matching and asserting `False`, throwing an exception, or printing an error, but never made type modifications to actually make these cases impossible.

Overall, participants' textual program edits often telegraphed information about their program design plan, and a particular subset of these edits signaled their intent to make future, related edits.

Hypothesis 4. Statically-typed functional programmers make certain types of program edits (signals) that reliably predict particular future program edits (responses).

5 EVALUATION

Having developed a grounded theory of how statically-typed functional programmers write code, we can now evaluate it by testing its predictions. The purpose of this evaluation is to provide increased confidence in our theory by approaching the problem with a second methodology; if our theory is correct, then the outcome of these quantitative experiments should be unsurprising, as they support conclusions already explained (in more detail and nuance) in the qualitative analysis.

[2]With thanks to Yaron Minsky for coining this phrase: https://blog.janestreet.com/effective-ml-revisited/.

We selected four hypotheses to test in quantitative lab experiments based on our subjective assessment of which would be most actionable for tool and language designers:

H1. When constructing new programs, statically-typed functional programmers iterate between editing types and editing expressions.
H2. Statically-typed functional programmers run their compilers on code they know will not compile.
H3. Statically-typed functional programmers face reduced workload when implementing a task with explicit pattern matches as compared to with combinators.
H4. Statically-typed functional programmers make certain types of program edits (signals) that reliably predict particular future program edits (responses).

Section 5.1 describes our method of testing these hypotheses, and Section 5.2 describes our results.

5.1 Method

Participants and recruitment. We conducted study sessions with 12 programmers who self-identified on a screening survey as 1) being comfortable with Haskell and 2) having at least two years of experience with Haskell. We recruited participants via Twitter and the /r/Haskell subreddit.

Study protocol. Study sessions consisted of one recorded 90-minute Zoom session during which we asked participants to complete four tasks (all done in the Haskell programming language to minimize between-language variability): a "workload" task aiming to compare the workload experience in two different programming styles, a "natural" task with no investigator interventions or questions, and two "compilation belief" tasks in which participants had to indicate whether or not they believed their code would successfully compile before using the compiler. We presented these tasks to participants as "Tasks 1–4" and always in the order listed above (which was especially important for ensuring a controlled experience between the two groups for the first task). The details of each task are described in the following sections alongside the hypotheses they test.

5.2 Results

For our experimental evaluation of our grounded theory, we opt for a conservative, nonparametric approach so as to avoid relying on assumptions about the measured distributions (e.g., that the survey scores discussed in Section 5.2.3 are normally distributed). Moreover, in this discussion, we use "population" to refer to the population of Haskell programmers with at least two years of experience. Our conclusions about whether our hypotheses are supported are naturally tied to the tasks we use to test them; while we attempted to make our lab tasks as realistic as possible, we invite the reader to consider the extent to which these claims may or may not generalize to real-world statically-typed functional programming practice beyond these specific tasks in Haskell. Lastly, we take as our significance level $\alpha = 0.05$.

5.2.1 Hypothesis 1: Iterating Between Type and Expression Construction. To operationalize **H1**, we analyzed data that we collected from the "natural" task, which we here describe in more detail:

> **Natural Task.** *Time limit: 30 min.* We asked participants to implement the game of Hangman, in which a player has to guess a secret word letter-by-letter before they run out of guesses. We requested that participants support features such as randomly selecting a word from a predefined list and an IO loop that indicates to the user the currently-guessed letters, how many guesses remained, whether they have won or lost, and if they have already guessed a letter. We informed participants that that they could use any combination of pattern matching, predefined functions (including those available via installing Haskell core libraries such as `random`), or anything else available to them in the language and any external resources.

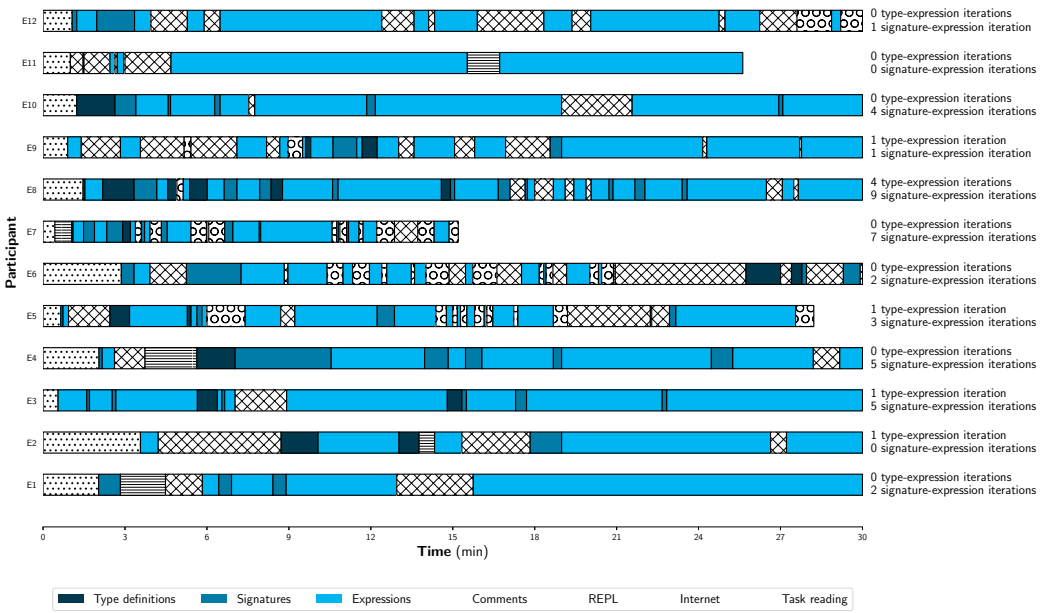

Fig. 3. **Timeline of observed programming activities.** 5/12 (≈ 42%) participants iterated between constructing types, then expressions, then types again; 11/12 (≈ 92%) participants iterated between constructing either types or signatures, then expressions, then either types or signatures again.

At each point in time, we observed which of the following seven categories of activity each participant was pursuing: 1) creating/modifying type definitions, 2) creating/modifying type signatures, 3) creating/modifying expressions, 4) creating/modifying comments, 5) using the REPL, 6) using the internet, or 7) reading the task. We depict participant timelines in Figure 3.

A strict interpretation of the notion of an *iteration* between type and expression editing can be defined as first editing type definitions, then editing expressions, then again editing type definitions; the number of such iterations for each participant is labeled on the right-hand vertical axis of Figure 3 as "type-expression iterations." In a general setting, this definition of iteration would need to be made more granular to account for programmers editing completely unrelated types at different points during the program authoring process, but since the Hangman task is relatively small (and thus results in types that are all closely related), this definition is sufficiently precise.

The proportion of participants who engaged in type-expression iteration was 5/12 (approximately 42%) with a Wilson confidence interval of (19%, 68%) at the 95% confidence level. These results indicate that there is strong reason to believe that at least 19% of the population would engage in type-expression iteration, even in the course of just 30 minutes for a Hangman game.

Relaxing the iteration criterion. We may also take a looser definition of "iterating between types and expressions" to include type *signatures* as well as type definitions. We can define a signature-expression iteration as first editing type signatures, then editing expressions, then again editing type signatures, a quantity which is also labeled on the right-hand vertical axis of Figure 3. The proportion of participants who engaged in type- or signature-expression iteration was 11/12 (approximately 92%) with a Wilson confidence interval of (65%, 99%) at the 95% confidence level. These results indicate that there is strong reason to believe that at least 65% of the population would engage in type- or signature-expression iteration.

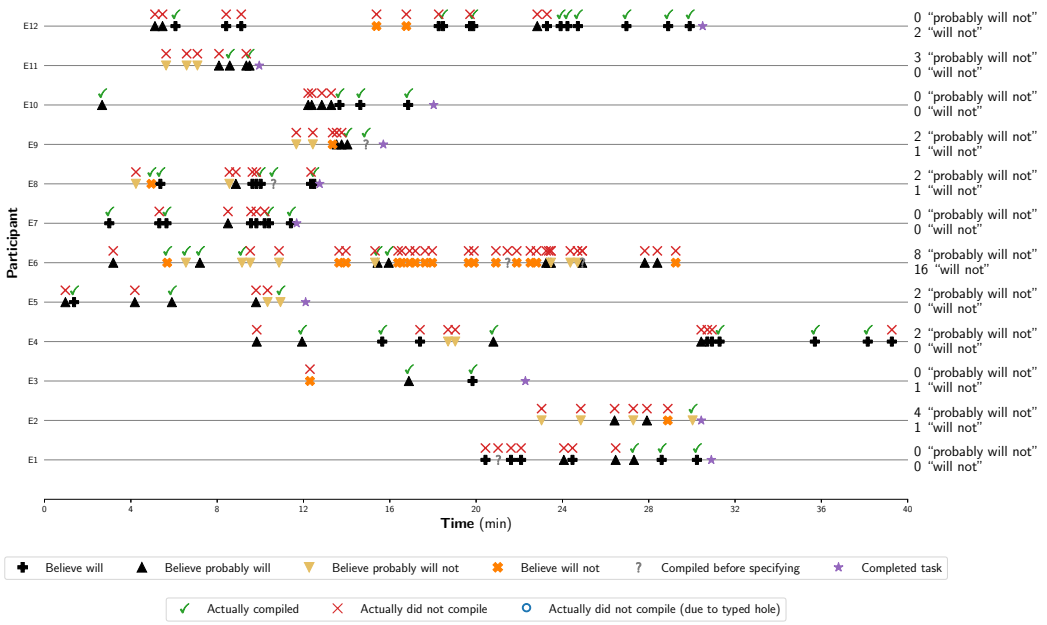

Fig. 4. **Program *drafting* phase: Will this code compile?** For Part 1 (drafting) of the compilation belief task, these timelines show every time a participant ran the compiler, whether they predicted the code (i) would, (ii) probably would, (iii) probably would not, or (iv) would not compile. Marks above the timeline indicate whether the code actually compiled. 6/12 (50%) participants ran their compiler immediately after indicating they knew their code would not compile.

5.2.2 Hypothesis 2: Expecting the Compiler to Fail. To operationalize **H2**, we analyzed data that we collected from the two "compilation belief" tasks, which we here describe in more detail:

> **Compilation Belief Task (Part 1).** *Time limit: 40 min.* In the first part of the compilation belief task, we asked participants to implement a key-value store where keys and values are both strings. We required that the store must support a main IO loop of `put` and `get` operations, with the twist that the `get` operation must take an integer parameter indicating the level of indirection. (For example, `get 2 key` should use the value stored in `key` as a lookup key for an additional lookup; in general, `get (n + 1) key store = get n (lookup key store) store`.) We informed participants that, as in the natural task, they could use anything available to them in the language as well as any external resources.
>
> We also asked participants to tell the investigator which of the following four statements was most applicable *before* they compiled (or refreshed their REPL) at any point in the task: (1) You know the code WILL compile; (2) You believe the code will PROBABLY compile; (3) You believe the code will PROBABLY NOT compile; (4) You know the code will NOT compile.

> **Compilation Belief Task (Part 2).** *Time limit: any time remaining in the 40 min allotted for Part 1 of the Compilation Belief Task.* We asked participants to refactor their key-value store program so that values were lists of strings rather than just strings. All the Part 1 rules applied, including the requirement that participants inform the investigator of their compilation belief before each compilation attempt.

Part 1. In Part 1 (Figure 4), the proportion of participants who ran their compiler with the "will NOT compile" belief was 6/12 (50%) with a Wilson confidence interval of (25%, 75%) at the 95% confidence level. These results indicate there is strong reason to believe that at least 25% of the

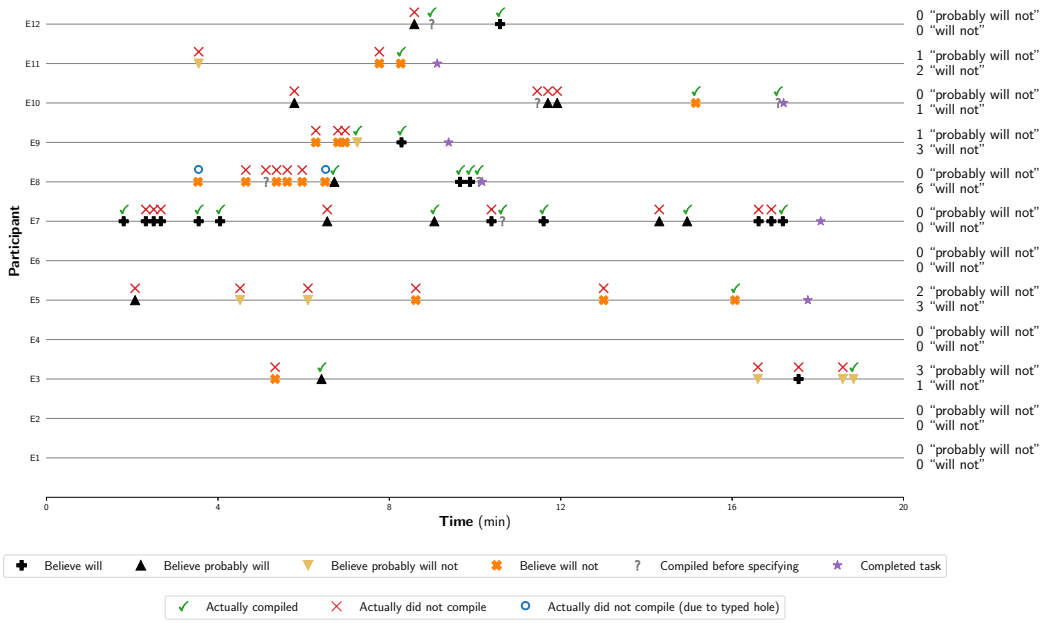

Fig. 5. **Program *refactoring* phase: Will this code compile?** For Part 2 (refactoring) of the compilation belief task, these timelines show every time a participant ran the compiler, whether they predicted the code (i) would, (ii) probably would, (iii) probably would not, or (iv) would not compile. Marks above the timeline indicate whether the code actually compiled. 6/8 (75%) participants ran their compiler immediately after indicating they knew their code would not compile.

population would run the compiler knowing their code will not compile, even just working on a well-defined 40-minute task.

Part 2. In Part 2 (Figure 5), among those who ran their compiler at all, the proportion of participants who ran their compiler with the "will NOT compile" belief was 6/8 (75%) with a Wilson confidence interval of (41%, 93%) at the 95% confidence level. These results are stronger than those in Part 1; one plausible reason is that Part 2 involves (approximately) refactoring code with a `Maybe String` into code with a `List (Maybe (List String))`—a challenging task for many participants, prompting many to use the compiler for guidance on constructing a well-typed expression. Overall, these results are consistent with the hypothesis that statically-typed functional programmers compile even when they believe compilation will fail.

Relaxing the belief criterion. We may also relax our definition of "expecting the compiler to fail" to include *any* negative belief about the compilation—in particular, including not just "will NOT" beliefs, but also "will PROBABLY NOT" beliefs. This change strengthens the statistical results for Part 1 considerably. (For Part 2, the results are exactly the same because the definitions happen to include precisely the same set of participants.)

In Part 1, the proportion of participants who ran their compiler with the "will NOT compile" or "will PROBABLY NOT" beliefs (Figure 4) was 9/12 (75%) with a Wilson confidence interval of (47%, 91%) at the 95% confidence level.

5.2.3 Hypothesis 3: Pattern Matching vs. Combinators. To test **H3**, we analyzed data that we collected from the "workload" task, which we here describe in more detail:

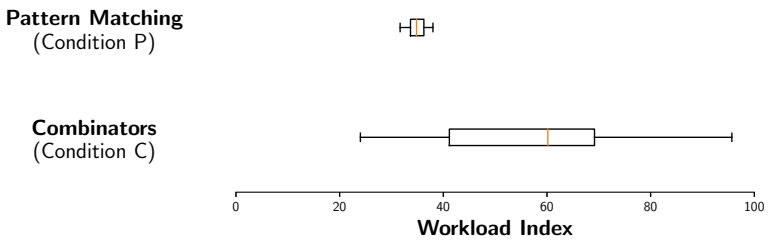

Fig. 6. **NASA-TLX measure of perceived workload.** The perceived workload is moderately lower for the condition in which participants could only use pattern matching vs. the condition in which they could only use combinators.

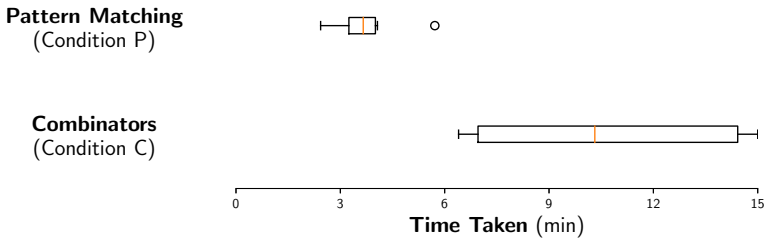

Fig. 7. **Time taken on task.** The median time for the pattern matching condition is significantly lower than the median time for the combinator condition; the slowest pattern matching participant completed the task faster than the fastest combinators participant.

Workload Task. *Time limit: 15 min (not including NASA-TLX survey).* We randomly assigned participants to one of two conditions, subject to the constraint that each condition have the same number of participants and as close to the same distribution of participant years of experience as was possible. In both conditions, we asked participants to implement a function `firstRight :: Maybe [Either a b] → Either String b` that returned the first `Right` value in the input list (or specialized errors if the input list was not present or had no `Right` values). In one condition—Condition P, for pattern matching—we instructed participants to use *only* pattern matching and recursion. In the other condition—Condition C, for combinators—we instructed participants to use *only* predefined Haskell functions and no recursion (that is, combinator style). In both conditions, we informed participants that they could use any external resources they wished.

After participants announced that they were satisfied with their solution to the task, we asked them to complete the weighted NASA-TLX (Task Load Index), a short, empirically-validated survey to measure their subjective experience of task workload (Hart and Staveland, 1988).

This task was designed to be as fair as possible to both conditions. The pattern matching condition cannot use functions such as `rights :: [Either a b] → [b]` that returns all right values in a list or `find : (a → Bool) → [a] → Maybe a` that searches a list based on a predicate. On the other hand, the combinator condition cannot use the opportunistic strategy of listing and handling each case one-by-one. The distributions of years of experience were approximately the same for the two conditions: [2, 2, 4, 5, 6, 18] for Condition P and [2, 3, 4, 4, 10, 11] for Condition C.

We plotted participants' NASA-TLX scores in Figure 6. Although time taken on task is not a measure of subjective workload, we collected this data to understand differences in time load, which we plotted in Figure 7. For the purposes of this experiment, we considered participants to have

Table 4. **Actions tracked in the natural task.** We operationalize *following the types* by considering the action Signature to be a signal and GloballyDefine to be a response. We operationalize *sketching* by considering ReferBeforeDefine to be a signal and GloballyDefine and LocallyDefine to be responses.

Action	Link	Short name
Referencing a name before defining it	The referenced name	ReferBeforeDefine
Creating/modifying a type signature	The name to which the signature applies	Signature
Defining a global (top-level) name	The defined name	GloballyDefine
Defining a local name	The defined name	LocallyDefine

completed a task when they explicitly declared that they were confident in their solution, as would be typical in their day-to-day programming; in all cases, the first author additionally observed that participants' solution did indeed meet the specification.

Compared to other statically-typed functional programming languages, the support for combinator-style programming in the Haskell standard library is relatively robust. The differences in workload between the experimental conditions may therefore be less pronounced than in the general statically-typed functional programming population.

Analysis of NASA-TLX scores. Descriptively, the median NASA-TLX score for Condition P was 34.8, and the median NASA-TLX score for Condition C was 60.2. The standard deviation in Condition P was 2.1, and the standard deviation in Condition C was 23.3, suggesting that Condition P participants perceived the workload to be approximately the same, whereas Condition C participants' perceived workloads varied substantially more (however, NASA-TLX scores may not be linearly correlated to perceived workload, which could affect the difference in standard deviation between the conditions).

Inferentially, let x_P denote the median NASA-TLX score in the population for Condition P and x_C for Condition C. We use Mood's two-tailed median test to test if the medians are the same, which yields $p = 0.083$ for $H_0 : x_P = x_C$ and $H_1 : x_P \neq x_C$. As $p > \alpha = 0.05$, we do not reject H_0; qualitatively, however, the trend of the data suggests that the pattern matching condition *may* be lower-workload than the combinators condition.

Analysis of time taken on task. The difference in time was greater than the difference in TLX scores. Descriptively, the median time taken on task for Condition P was 3.7 minutes, and the median time taken on task for Condition C was 10.3 minutes. Moreover, the slowest participant in Condition P completed their task faster than the fastest participant in Condition C. As with the NASA-TLX scores, the standard deviation in time taken between the two conditions was substantially different: 1.0 minutes for Condition P and 3.7 minutes for Condition C.

Inferentially, let x_P denote the median time taken on task for Condition P and x_C for Condition C. We again use Mood's two-tailed median test to test if the medians are the same, which yields $p = 0.004$ for $H_0 : x_P = x_C$ and $H_1 : x_P \neq x_C$. As $p \ll \alpha = 0.05$, this result provides strong evidence to reject the null hypothesis, which indicates that completing the task with pattern matching is faster than completing it with combinators.

Qualitative observations. In addition to our quantitative results, we include two observations:

1. All participants in Condition C used either Hoogle (a tool that looks up Haskell functions by name or type signature) or the "lookup type" command in the Haskell REPL.
2. Some participants in Condition C first constructed parts of their solution using pattern matching, then switched over to a combinator style. In particular, one participant wrote their *entire solution* using pattern matching, then later transferred it all to a combinator style.

Following the Types

	No Signal	Signal
No Response	—	11
Response	32	41

Sketching

	No Signal	Signal
No Response	—	5
Response	124	22

Fig. 8. **Signal and response frequency.** *Predictivity* is the bottom-right cell divided by the sum of the right column (Following the Types: $\frac{41}{11+41} \approx 79\%$; Sketching: $\frac{22}{5+22} \approx 81\%$). *Commonness* is the bottom-right cell divided by the sum of the bottom row (Following the Types: $\frac{41}{32+41} \approx 56\%$; Sketching: $\frac{22}{124+22} \approx 15\%$).

5.2.4 Hypothesis 4: Signaling Future Edits. To assess **H4**, we analyzed data we collected from the "natural" task, which we described in Section 5.2.1. Specifically, we recorded each time a participant performed one of the four *actions* listed in Table 4. Each time a participant performed one of these actions, we also recorded some additional information in the form of a *link*, also listed in Table 4. Note that each link can occur for a given action at most once.

For the purposes of this experiment, we are interested in viewing subsets of these actions as *signals* and subsets of these actions as *responses*. Based on frequencies in our data, we aim to understand whether signals reliably predict responses and whether signals reliably precede responses. More specifically, we are interested in estimating:

$$P(\textbf{response} \text{ with link } \ell \text{ occurs} \mid \textbf{signal} \text{ with link } \ell \text{ occurs}) \qquad (\textit{predictivity})$$
$$P(\textbf{signal} \text{ with link } \ell \text{ occurs} \mid \textbf{response} \text{ with link } \ell \text{ occurs}) \qquad (\textit{commonness})$$

The links are critical. It makes little sense to predict whether programmers will define a global variable x after writing a signature that applies to y—there is no reason to believe that these will be meaningfully correlated or will produce actionable predictions for tooling.

We calculated the *empirical predictivity* and *empirical commonness* as relative frequencies in our observed samples as estimators for the corresponding true probabilities for two practices modeled as signal-response pairs: (i) *following the types* and (ii) *sketching*. Figure 8 provides the raw counts of the actions observed across all 12 participants. We investigated these particular signal-response pairs because we expected (based on the grounded theory) that they would be highly predictive and thus useful to tool designers; we speculate that future work using automated analysis of large-scale, fine-grained program edits could reveal many additional signal-response pairs.

Following the types: Signaling global definitions with type signatures. For this analysis, we take the set of signals to be {Signature} and the set of actions to be {GloballyDefine}. This setup encodes the notion of *following the types* (Section 4.2.2) because, when following the types, programmers define a type signature first, then let their implementation flow from it; this is idiomatic for global definitions because these are the definitions that typically require nontrivial type-based decomposition and planning. In this setup, the empirical predictivity across all participants and labels was 79% with a Wilson confidence interval of (66%, 88%) at the 95% confidence level, and the empirical commonness was 56% with a Wilson confidence interval of (45%, 67%) at the 95% confidence level. These results indicate that creating or modifying a type signature often *does* signal that the programmer will soon define a corresponding global name (but not always, because participants sometimes changed their minds and deleted these signatures), and, perhaps more interestingly, a new global name definition often *is* preceded by a linked type signature change.

Sketching: Signaling definitions with undefined references. For this analysis, we take the set of signals to be {ReferBeforeDefine} and the set of actions to be {GloballyDefine, LocallyDefine}. This setup operationalizes a restricted notion of *sketching* (Section 4.2.3) because, when sketching,

programmers implement a program skeleton with expression holes (commonly undefined references) which are later filled (defined). In this setup, the empirical predictivity across all participants and labels was 81% with a Wilson confidence interval of (63%, 92%) at the 95% confidence level, and the empirical commonness was 15% with a Wilson confidence interval of (10%, 22%) at the 95% confidence level. Although 15% is relatively low, it is worth contextualizing this number by noting that 15% of *all names defined in the task* were first signaled via sketching. These results indicate that referring to an undefined name often *does* signal that the programmer will soon define that name, but that new definitions are only *sometimes* preceded by a reference to the yet-to-be-defined name.

6 IMPLICATIONS AND UNRESOLVED QUESTIONS

We now discuss the implications our findings have for tooling and language design, as well as some further questions that remain unresolved.

6.1 Iterating Between Type and Expression Construction

H1 indicates the need for tools that reduce the viscosity (Green and Petre, 1996)—or, resistance to change—of types during development. Moreover, tools should support cyclic changes in programmer mindset between working at the type level and at the expression level. For example, tools could leverage the pre-existing pattern of type-expression iteration by using program repair techniques to provide an interface to 1) modify types and automatically preview corresponding expression changes or 2) modify expressions and automatically preview corresponding type changes. Such a tool could support the exploratory type construction phase we observed in this study.

More generally, tool designers should consider viewing types as amenable to change; if a program does not type check, it may not be the expressions that are at fault, but the types or domain modeling. Automatically suggesting alternative types in this case could support the iterative type-expression cycle and help guide users to correct their mental models of the problem domain.

6.2 Expecting the Compiler to Fail

There are three immediate implications of **H2**. First, statically-typed functional programmers use their compiler for reasons other than producing a compiled program (some of which are described in Section 4.2.1). Second, statically-typed functional programmers often want insights from automated tooling *before* their programs are compilable, suggesting that automated tooling should do its best to extract as much information as possible from code that does not compile. And, third, compiler error messages may not really be caused by programmer "errors" at all, but by deliberate behavior intended to elicit compiler feedback. Thus, while prior work has investigated how compilers should communicate errors to programmers (Barik et al., 2018, Becker et al., 2018, Nienaltowski et al., 2008), reimagining compiler feedback as sometimes being *directive* in nature rather than strictly *corrective* could align it more closely to statically-typed functional programmers' needs.

Unresolved Question. Our test of **H2** does not explain *why* statically-typed functional programmers run their compilers on code they expect not to compile. What hypotheses do they have before running their compiler, and exactly what information do they seek to obtain? Section 4.2.1 offers some insight, but future work could produce a focused exploration of this question.

6.3 Decomposing Tasks with Pattern Matching

Our test of **H3** does not explain *why* statically-typed functional programmers might experience less mental and temporal workload using pattern matching; it may be that pattern matching offers an explicit, textual representation of sub-tasks, that it forces programmers to deal with recursion

explicitly, that it is a uniform interface for dealing with nearly all data structures in the language, or something else entirely.

Unresolved Question. Why might pattern matching incur reduced workload compared to combinators?

However, our **H3** results—combined with the fact that some statically-typed functional programmers prefer that their implementation eventually be streamlined to use combinators (Section 4.3.1)—does indicate an opportunity for tooling to handle cases where programmers prefer to draft a program in one style but prefer to *maintain* a program in a higher-workload style. An example of such a tool in this domain could take a datatype as input, prompt the programmer to opportunistically fill out whatever cases they can, then return a pipeline of combinators that are equivalent (or generalize the provided cases if they are not exhaustive).

We can frame the low-workload code to high-workload code transformation task as a synthesis problem similar to verified lifting (Kamil et al., 2016): given an opportunistically-constructed (possibly partial) program P and a set of subjectively desirable or streamlined components C, find a program P' equivalent to P on its inputs using only components from C. This perspective hints at a general strategy for tool need-finding: identify tasks that result in the creation of expressions with similar semantics but that require different workloads to achieve, then try to automatically transfer solutions of the easier task to the harder. This opportunity exists in any case where we observe substitutable styles in which one is lower-workload but another is a more desirable end product.

Unresolved Question. What other common tasks result in programs with similar semantics but exhibit workload differentials that could be exploited by automated tooling?

This finding also has implications for language designers: it is worthwhile to consider which language constructs admit low-workload or opportunistic construction, and how such constructs relate to what a community considers desirable code.

6.4 Signaling Future Edits

To test **H4**, we analyzed predictivity and commonness of signal-response pairs. But are these metrics useful? What do they tell us about a tool that leverages a signal-response pair? In short:

- High predictivity indicates that such a tool would anticipate user actions well, since programmers often follow the signal with the response.
- High commonness indicates that such a tool will have a plausible audience, since programmers already use the signal before the response.

Predictive and common signal-response pairs thus offer fertile ground for automated tooling.

Unresolved Question. What predictive and common signal-response editing pairs exist in the statically-typed functional programming community? We identified two such pairs—roughly, following the types and sketching—but a larger-scale study could instrument code editors to automatically detect and cluster actions into predictive and common signal-response pairs.

The particular signal-response pairs we identified suggest two specific tool categories:

Tools for following the types. Given that (at least in Haskell) statically-typed functional programmers often already write type signatures for global definitions in advance, tooling that relies on having a type signature before implementation will likely not overburden programmers. Tools that exploit this pattern might, for instance, automate aspects of type-directed decomposition.

Tools for sketching. A program sketch provides a rich source of information about yet-to-be-defined functions, such as inferred types and usage patterns. Statically-typed functional programmers already 1) write sketches with holes and 2) later fill them; this pattern suggests an opportunity for tooling that supports or automates either of these processes.

7 LIMITATIONS

A fundamental limitation of the qualitative portion of this work is that asking participants to design *and* think aloud (as we did) can overload their working memory (Davies and Castell, 1994). The grounded theory sessions may thus be skewed toward more opportunistic behavior than is typical—although such working memory overload may also be common in some working environments. Participants did *not* have to think aloud for the experimental portion of this study, but our experimental setup suffers from the limitation of taking place in a controlled environment; the tasks were held constant and may or may not be ecologically valid. This same limitation applies to the development of the grounded theory but is partially mitigated by leaving the tasks open-ended and including livestreamed programming sessions in the analysis.

8 RELATED WORK

Throughout Section 4, we discussed work that is particularly relevant to our individual findings; in this section, we survey work that is related to the overall approach and goals of our study.

8.1 Related Methods

To our knowledge, we have conducted the first grounded theory analysis of narrated programming sessions by expert programmers. However, we are not the first to study programmers via grounded theory or observation.

8.1.1 Grounded Theory in Software Engineering. The grounded theory approach to data analysis has become increasingly popular in the software engineering community (Adolph et al., 2012, Coleman and O'Connor, 2007, Hoda et al., 2013, Jantunen and Gause, 2014, Pang et al., 2020, Zhang et al., 2020). Researchers have applied GTM to topics ranging from how API designers gather and interpret user feedback (Zhang et al., 2020) to how DevOps education functions (Pang et al., 2020). By far the most common approach is to use interviews as the sole data for the grounded theory. Including programming sessions is much less common, even though interviewees are often programmers. Sedano et al. (2017) describes a notable exception, a GTM analysis of software engineering practices, including participant observation in the form of pair programming (not necessarily including thinking aloud); the authors stress observations were significantly more informative than interviews alone.

8.1.2 Contextual Inquiry and Observational Research of Programmers. Contextual inquiry is a data collection research method in which investigators study a group of people by acting as an *apprentice* and observing the behavior of participants (the experts) as they go about their day-to-day activities (Beyer and Holtzblatt, 1997). For example, Ko (2003) observes expert event-based programmers to understand how they write, test, and debug code, and Samuel (2009) studies how Java programmers working on mobile applications interact with their IDEs. Most of the observational research in the programming space is not aimed at building a theory but rather at answering specific, pre-defined research questions, so their analytic methods (e.g., closed coding) are very different from the open-ended grounded theory approach we used. Despite the difference in analytic approaches, our data collection process builds on the methods in this related line of research, turning observations of programming session into insights about programmer processes.

8.1.3 Think-Aloud Studies of Programming. To learn more about the mental processes by which programmers arrive at their authoring decisions, researchers can ask participants to think aloud as they write code. For example, Whalley and Kasto (2014) and Fisler and Castro (2017) use this think-aloud technique to understand novice programmers' authorship strategies. The protocol for our own think-aloud study is influenced by these works, but these studies focus specifically on novice programmers and employ narrative analyses (Riessman, 1993) rather GTM.

8.1.4 Theory Evaluation. Evaluating behavioral theories of software engineering—e.g., testing the falsifiable hypotheses they generate—both builds confidence in a theory and often produces actionable insights for tool designers, language designers, and computer science educators. For example, Ko and Myers (2003) develop and evaluate a theory of programming errors, and Thayer et al. (2021) do the same for a theory of API knowledge. Although neither of these theories arose from a grounded theory analysis, our approach to empirical evaluation is similar to theirs.

8.2 Related Goals

As the goal of this study is to understand how statically-typed functional programmers write code, we now discuss prior work on understanding how various populations of programmers write code.

8.2.1 Psychology of Programming. Among other goals, studies of the psychology of programming (PoP) aim to understand the cognitive processes behind programmer authoring patterns. Although our work is concerned more with the authoring patterns than the underlying cognitive processes, we build on a foundation of related studies in the psychology of programming space.

A long-running line of PoP research investigates the *strategies* that programmers use to write code. Early studies (Farrell et al., 1984, Pirolli and Anderson, 1985, Spohrer and Soloway, 1989) found that novices base new programs on known solutions, tweaking a familiar solution to fit the new problem. Spohrer and Soloway (1986) find that many novice programming errors arise from *plan composition* challenges. Programming *plans* (Soloway and Ehrlich, 1986) are a cognitively plausible element of programmer knowledge—specifically, program fragments that together accomplish a common programming task, such as the running sum loop plan. Novices combining plans struggle with issues such as letting previously used plans pollute a related plan, cognitive load and working memory overload, and unexpected boundary cases (Spohrer and Soloway, 1986). Overall, the strategies literature finds "experts seem to acquire a collection of strategies for performing programming tasks, and these may determine success more than does the programmer's available knowledge" (Gilmore, 1990). Recent work like LaToza et al. (2020)'s demonstrates the power of PoP work for tool design, describing a tool that leverages the PoP strategies theory to make programmers objectively more successful at design and debugging tasks.

As discussed in Section 2, the strategies mentioned above that programmers employ can be roughly divided into two categories: hierarchical and opportunistic. Jeffries et al. (1981) contributed the first study of top-down programming strategies, soon followed by Adelson and Soloway (1985), Anderson et al. (1984), Pirolli (1986) and Pirolli and Anderson (1985). Visser (1987) contributed the first work studying how programmers shift between top-down and bottom-up strategies, soon followed by Guindon et al. (1987). Although top-down, hierarchical patterns of program construction are often considered paradigmatic of expert programmers, Détienne (2002, Chapter 3) observes "the hierarchical structure reflects the result of the activity but not the organization of the activity itself" and states even experts engage in opportunistic strategies, a pattern we also observed in our study (Section 4.3.1). More broadly, top-down and bottom-up strategies surface across many programming tasks, in for example, software comprehension (Roehm et al., 2012) and understanding software evolution (von Mayrhauser and Vans, 1995).

Most closely related to our study is Castro and Fisler (2016)'s investigation of the interplay of hierarchical and opportunistic thinking in novice functional programmers, which found that students who engaged in on-the-fly hierarchical task decomposition made plan-composition errors. Similarly, Castro and Fisler (2020) describe how novice functional programmers iterate between reasoning about task decomposition and code implementation. These works focus on patterns in novices' programming practices, but our discussion of programming as a process of clarification (Section 4.3.2) identifies similar patterns at play in expert programmers.

Aside from the interplay of hierarchical and opportunistic strategies, the most relevant works from PoP touch on how attention affects programming tasks. Like our observations about focusing techniques, this line of research identifies patterns in how programmers direct their attention. Most of this work focuses specifically on debugging (Bednarik and Tukiainen, 2004, Hejmady and Narayanan, 2012, Lewis, 2012), program comprehension (Bednarik and Tukiainen, 2006, Guéhéneuc, 2006, Letovsky, 1987, Yusuf et al., 2007), and code summarization (Rodeghero et al., 2015, Rodeghero and McMillan, 2015, Rodeghero et al., 2014) rather than program authoring, but some findings overlap with our own. Notably, Lewis (2012)'s hypothesis that debugging success relies on the skill of identifying what elements of program state are important resonates with our participants' remarks in Section 4.2.2 about the benefit of restricting attention to particular subparts of a program.

8.2.2 Natural Programming. Myers et al. (2004) put forth a research agenda of *natural programming*, or "aiming for the language and the environment to work the way nonprogrammers expect." *Natural programming elicitation* studies involve asking non-programmers or novice programmers to express programs with the concepts and abstractions they find most natural (Myers et al., 2016). Researchers have used natural programming elicitation to, for example, understand how untrained users specify boolean queries (Pane and Myers, 2000) or how novice programmers express programming concepts (Myers et al., 2004), and the natural programming approach more generally to, for example, improve API learnability (Stylos and Myers, 2008). Our study shares the goal of understanding programmers' intuitions but differs in its treatment of pre-existing experience; we are concerned with the existing authoring patterns that programmers deploy when they are already comfortable with statically-typed functional programming, regardless of the inherent naturalness of these patterns. A natural programming elicitation study with statically-typed functional programmers would serve as an interesting comparison of an "ideal" world to the world as it is now.

8.2.3 Collections of Program Authorship Patterns. A few lines of prior work compile lists of programmer authoring patterns. For example, *design patterns* are reusable high-level program design templates popularized by the influential *Design Patterns* compendium Gamma et al. (1995). Although Beck et al. (1996) report on the success of design patterns in industry for promoting clear communication and reusable code, design patterns offer little insight into how these patterns are achieved in the first place. In this study, we are concerned not just with patterns in programmers' final programs, but with the processes by which they achieve those programs.

Collections of *automated program transformations* ubiquitous in modern development environments (such as Introduce Parameter and Move Definition) also offer plausible lists of programmer behaviors. Thompson (2004) details many such transformations applicable specifically to statically-typed functional programming languages, but, while these transformations provide a detailed characterization of some of the semantics-preserving low-level edits programmers undertake, they do not capture the high-level intent of programmers as they construct code.

8.2.4 Program Authorship Patterns of Statically-Typed Functional Programmers. Lubin and Chugh (2019) lay out a vision of studying common code authoring patterns of statically-typed functional programmers, and Lubin (2021) presents early data from the grounded theory presented in Section 4.

9 CONCLUSION

This study deepens our understanding of how statically-typed functional programmers write code. Our mixed-methods approach provides both a qualitative account of the statically-typed functional programming process to capture its nuances as well as a quantitative validation to test our theory's predictions. Future work can build on this knowledge to design evidence-based tools that leverage existing program authoring patterns. Our results suggest that this same methodology—grounded theory of think-aloud programming sessions with semi-structured interviews, followed by an experimental validation of the developed theory—could be applied to other programming populations where we currently lack need-finding analyses, ultimately leading to more accessible and expressive tooling for understudied communities.

ACKNOWLEDGMENTS

We are indebted to the anonymous participants of these studies for making this research possible. We are also grateful to the anonymous OOPSLA '21 reviewers and CHI SRC '20 judges for their feedback. The first author additionally thanks Ravi Chugh for the insightful conversations that led to the development of some of the questions addressed by this study. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1752814 and is supported in part by NSF Grants CA-HDR 2029457, CA-HDR 1936731, and CA-HDR 2033558 as well as by gifts from Apple and Google.

REFERENCES

Beth Adelson and Elliot Soloway. 1985. The Role of Domain Experience in Software Design. *Transactions on Software Engineering* (Nov. 1985). https://doi.org/10.1109/TSE.1985.231883

Steve Adolph, Philippe Kruchten, and Wendy Hall. 2012. Reconciling Perspectives: A Grounded Theory of How People Manage the Process of Software Development. *Journal of Systems and Software* (June 2012). https://doi.org/10.1016/j.jss.2012.01.059

John R. Anderson, Robert Farrell, and Ron Sauers. 1984. Learning to Program in LISP. *Cognitive Science* (April 1984). https://doi.org/10.1016/S0364-0213(84)80013-0

Jon Awbrey and Susan Awbrey. 1995. Interpretation as Action: The Risk of Inquiry. *Inquiry: Critical Thinking Across the Disciplines* 1 (Feb. 1995). https://doi.org/10.5840/inquiryctnews199515125

Alan Baddeley. 2007. *Working Memory, Thought, and Action.* https://doi.org/10.1093/acprof:oso/9780198528012.001.0001

Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers? In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* https://doi.org/10.1145/3236024.3236040

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (Nov. 2020). https://doi.org/10.1145/3428295

Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O. Coplien, Lutz Dominick, and Frances Paulisch. 1996. Industrial Experience with Design Patterns. In *International Conference on Software Engineering (ICSE).* https://doi.org/10.1109/ICSE.1996.493406

Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *SIGCSE Technical Symposium.* https://doi.org/10.1145/3159450.3159453

Roman Bednarik and Markku Tukiainen. 2004. Visual Attention and Representation Switching in Java Program Debugging: A Study Using Eye Movement Tracking. In *Psychology of Programming Interest Group (PPIG).*

Roman Bednarik and Markku Tukiainen. 2006. An Eye-Tracking Methodology for Characterizing Program Comprehension Processes. In *Symposium on Eye Tracking Research & Applications (ETRA).* https://doi.org/10.1145/1117309.1117356

Hugh Beyer and Karen Holtzblatt. 1997. *Contextual Design: Defining Customer-Centered Systems.* Morgan Kaufmann Publishers Inc.

Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *SIGCSE Technical Symposium.* https://doi.org/10.1145/2839509.2844574

Francisco Enrique Vicente Castro and Kathi Fisler. 2020. Qualitative Analyses of Movements Between Task-Level and Code-Level Thinking of Novice Programmers. In *SIGCSE Technical Symposium.* https://doi.org/10.1145/3328778.3366847

Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis.* SAGE Publishing Inc.

Gerry Coleman and Rory O'Connor. 2007. Using Grounded Theory to Understand Software Process Improvement: A Study of Irish Software Product Companies. *Information and Software Technology* (June 2007). https://doi.org/10.1016/j.infsof.2007.02.011

Simon P. Davies. 1991. Characterizing the Program Design Activity: Neither Strictly Top-down nor Globally Opportunistic. *Behaviour & Information Technology* (May 1991). https://doi.org/10.1080/01449299108924281

Simon P. Davies and Adrian M. Castell. 1994. From Individuals to Groups Through Artifacts: The Changing Semantics of Design in Software Development. In *User-Centred Requirements for Software Engineering Environments*, David J. Gilmore, Russel L. Winder, and Françoise Détienne (Eds.). https://doi.org/10.1007/978-3-662-03035-6_2

Françoise Détienne. 2002. *Software Design — Cognitive Aspect.* https://doi.org/10.1007/978-1-4471-0111-6

Robert G. Farrell, John R. Anderson, and Peter L. Pirolli. 1984. Learning to Program Recursion. In *Proceedings of the Sixth Annual Cognitive Science Meetings.*

Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *International Conference on Computing Education Research (ICER).* https://doi.org/10.1145/3105726.3106183

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

David J. Gilmore. 1990. Expert Programming Knowledge: A Strategic Approach. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). https://doi.org/10.1016/B978-0-12-350772-3.50019-7

Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *International Symposium on Haskell (Haskell).* https://doi.org/10.1145/3242744.3242760

Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine de Gruyter.

Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* (June 1996). https://doi.org/10.1006/jvlc.1996.0009

Yann-Gaël Guéhéneuc. 2006. TAUPE: Towards Understanding Program Comprehension. In *Conference of the Center for Advanced Studies on Collaborative Research (CASCON).* https://doi.org/10.1145/1188966.1188968

Raymonde Guindon. 1990. Designing the Design Process: Exploiting Opportunistic Thoughts. *Human–Computer Interaction* (1990). https://doi.org/10.1080/07370024.1990.9667157

Raymonde Guindon, Herb Krasner, and Bill Curtis. 1987. Breakdowns and Processes during the Early Activities of Software Design by Professionals. In *Empirical Studies of Programmers: Second Workshop.*

Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Advances in Psychology*, Peter A. Hancock and Najmedin Meshkati (Eds.). https://doi.org/10.1016/S0166-4115(08)62386-9

Prateek Hejmady and N. Hari Narayanan. 2012. Visual Attention Patterns during Program Debugging with an IDE. In *Symposium on Eye Tracking Research and Applications (ETRA).* https://doi.org/10.1145/2168556.2168592

Rashina Hoda, James Noble, and Stuart Marshall. 2013. Self-Organizing Roles on Agile Software Development Teams. *Transactions on Software Engineering* 3 (March 2013). https://doi.org/10.1109/TSE.2012.30

Sami Jantunen and Donald C. Gause. 2014. Using a Grounded Theory Approach for Exploring Software Product Management Challenges. *Journal of Systems and Software* (Sept. 2014). https://doi.org/10.1016/j.jss.2014.03.050

Robin Jeffries, Althea A. Turner, Peter G. Polson, and Michael E. Atwood. 1981. The Processes Involved in Designing Software. In *Cognitive Skills and Their Acquisition* (1st ed.).

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/2908080.2908117

Amy J. Ko. 2003. A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment. In *Extended Abstracts on Human Factors in Computing Systems (CHI EA).* https://doi.org/10.1145/765891.766135

Amy J. Ko and Brad A. Myers. 2003. Development and Evaluation of a Model of Programming Errors. In *Symposium on Human Centric Computing Languages and Environments.* https://doi.org/10.1109/HCC.2003.1260196

Thomas D. LaToza, Maryam Arab, Dastyni Loksa, and Amy J. Ko. 2020. Explicit Programming Strategies. *Empirical Software Engineering* (July 2020). https://doi.org/10.1007/s10664-020-09810-1

Stanley Letovsky. 1987. Cognitive Processes in Program Comprehension. *Journal of Systems and Software* (Dec. 1987). https://doi.org/10.1016/0164-1212(87)90032-X

Colleen M. Lewis. 2012. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. In *International Conference on International Computing Education Research (ICER).* https://doi.org/10.1145/2361276.2361301

Justin Lubin. 2021. How Statically-Typed Functional Programmers Author Code. In *Extended Abstracts on Human Factors in Computing Systems.* https://doi.org/10.1145/3411763.3451515

Justin Lubin and Ravi Chugh. 2019. Type-Directed Program Transformations for the Working Functional Programmer. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Sarah Chasins, Elena L. Glassman,

and Joshua Sunshine (Eds.). https://doi.org/10.4230/OASIcs.PLATEAU.2019.3

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue ICFP* (Aug. 2020). https://doi.org/10.1145/3408991

Michael Muller. 2014. Curiosity, Creativity, and Surprise as Analytic Tools: Grounded Theory Method. In *Ways of Knowing in HCI*, Judith S. Olson and Wendy A. Kellogg (Eds.). https://doi.org/10.1007/978-1-4939-0378-8_2

Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* (2016). https://doi.org/10.1109/MC.2016.200

Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Communications of the ACM* (2004). https://doi.org/10.1145/1015864.1015888

Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices? In *SIGCSE Technical Symposium*. https://doi.org/10.1145/1352135.1352192

John F. Pane and Brad A. Myers. 2000. Tabular and Textual Methods for Selecting Objects from a Group. In *International Symposium on Visual Languages (VL/HCC)*. https://doi.org/10.1109/VL.2000.874379

Candy Pang, Abram Hindle, and Denilson Barbosa. 2020. Understanding Devops Education with Grounded Theory. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. https://doi.org/10.1145/3377814.3381711

Peter Pirolli. 1986. A Cognitive Model and Computer Tutor for Programming Recursion. *Human–Computer Interaction* 4 (Dec. 1986). https://doi.org/10.1207/s15327051hci0204_3

Peter L. Pirolli and John R. Anderson. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology* (1985). https://doi.org/10.1037/h0080061

Catherine Kohler Riessman. 1993. *Narrative Analysis*. SAGE Publishing Inc.

Paige Rodeghero, Cheng Liu, Paul W. McBurney, and Collin McMillan. 2015. An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization. *Transactions on Software Engineering* (Nov. 2015). https://doi.org/10.1109/TSE.2015.2442238

Paige Rodeghero and Collin McMillan. 2015. An Empirical Study on the Patterns of Eye Movement during Summarization Tasks. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. https://doi.org/10.1109/ESEM.2015.7321188

Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568247

Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How Do Professional Developers Comprehend Software? In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2012.6227188

Ogunleye O. Samuel. 2009. MobiNET: A Framework for Supporting Java Mobile Application Developers through Contextual Inquiry. In *International Conference on Adaptive Science Technology (ICAST)*. https://doi.org/10.1109/ICASTECH.2009.5409746

Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Lessons Learned from an Extended Participant Observation Grounded Theory Study. In *International Workshop on Conducting Empirical Studies in Industry (CESI)*. https://doi.org/10.1109/CESI.2017.2

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. *ACM SIGPLAN Notices* (June 2005). https://doi.org/10.1145/1064978.1065045

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1168857.1168907

Elliot Soloway and Kate Ehrlich. 1986. Empirical Studies of Programming Knowledge. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). https://doi.org/10.1016/B978-0-934613-12-5.50042-2

James C. Spohrer and Elliot Soloway. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM* (July 1986). https://doi.org/10.1145/6138.6145

James C. Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884833

Jeffrey Stylos and Brad A. Myers. 2008. The Implications of Method Placement on API Learnability. In *International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/1453101.1453117

Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. 2021. A Theory of Robust API Knowledge. *Transactions on Computing Education* (Jan. 2021). https://doi.org/10.1145/3444945

Simon Thompson. 2004. Refactoring Functional Programs. In *Advanced Functional Programming (AFP)*. https://doi.org/10.1007/11546382_9

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *ACM SIGPLAN Notices* (June 2014). https://doi.org/10.1145/2666356.2594340

Cathy Urquhart and Walter Fernandez. 2006. Grounded Theory Method: The Researcher as Blank Slate and Other Myths. In *International Conference on Information Systemcs (ICIS)*. https://aisel.aisnet.org/icis2006/31

Willemien Visser. 1987. Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer. In *Empirical Studies of Programmers*, G. Olson, S. Sheppard, and E. Soloway (Eds.).

Anneliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* (Aug. 1995). https://doi.org/10.1109/2.402076

Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *Conference on Innovation & Technology in Computer Science Education (ITiCSE)*. https://doi.org/10.1145/2591708.2591762

Bin Xie and Gavriel Salvendy. 2000. Review and Reappraisal of Modelling and Predicting Mental Workload in Single- and Multi-Task Environments. *Work & Stress* (2000). https://doi.org/10.1080/026783700417249

Shehnaaz Yusuf, Huzefa Kagdi, and Jonathan I. Maletic. 2007. Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2007.10

Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. 2020. Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study. In *Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3313831.3376382