CSE 291 I: Usability of Programming Languages ("Programmers Are People Too")

Michael Coblenz

Justin Lubin and Sarah E. Chasins. 2021. How Statically-Typed Functional Programmers Write Code. OOPSLA.

How Do Statically-Typed Functional Programmers Write Code?

- Like studying a tribe with which we aren't familiar
- 15 participants
  - 45 minutes: think-aloud while doing given tasks
  - 45 minutes: semi-structured interview
- 15 livestreamed programming videos (≈ 1 hour each)
  - Goal: more diverse, more realistic tasks

Methods work great together!

Tasks

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).

Semi-Structured Interview?

- "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"

Grounded Theory Process (From Paper, 2.1)

- In video: 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."
- Grouped descriptions into about 100 summary tags that group similar activities. The above two tags were grouped under "switches from explicit case to pipeline."
- 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.")
- 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" … This tag was eventually refactored to "interplay of hierarchical and opportunistic programming."
- Reviewed theory to derive hypotheses to test in future work (and in quantitative evaluation in Section 5).

Theory

- Participants often started a task by iteratively constructing types to model their problem domain and encode design decisions.
- 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.
- 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.
- 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.

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. "

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)"
- Compilers as corrective tools vs. as directive tools
  - Directive: error messages served as a to-do list
  - Hypothesis: When statically-typed functional programmers refactor a type definition, they use the compiler error messages as a to-do list.

Hierarchical and Opportunistic Programming

- Hierarchical: type-guided programming
- Opportunistic: bottom-up construction, repeatedly wrapping expressions
  - Some people used the top level of their file as a notepad
  - Pattern-matching to narrow the problem down and analyze cases separately.
- Hypothesis 3. Statically-typed functional programmers face reduced workload when implementing a task with explicit pattern matches as compared to with combinators.

Diversity of Reasoning Approaches

- P8 introduced local, argument- passing state to their calculator in Elm to implement history
- 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).

Evaluating the Theory

- Are we done? That's the theory?
- If the theory is valid, it should make accurate predictions.
- How would YOU test the theory?

How Should We Test?

- Participants often started a task by iteratively constructing types to model their problem domain and encode design decisions.
- 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.
- 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.
- 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.

Hypotheses

- 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).

Hypothesis Tests

- "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. "

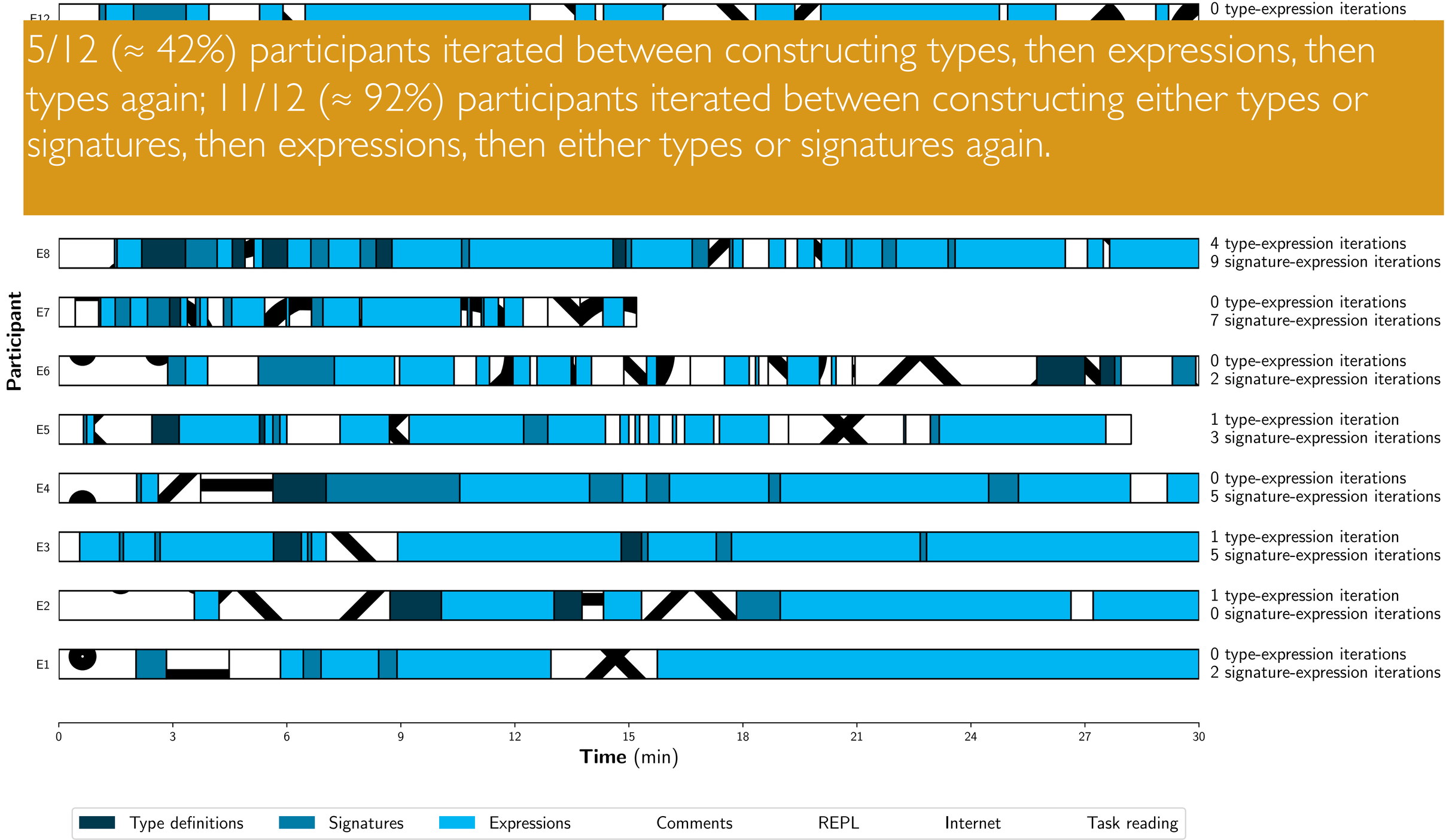
- "Study protocol. Study sessions consisted of one recorded 90-minute Zoom session during which we asked participants to complete four tasks (all done in the Haskell programming language to minimize between-language variability): a 'workload' task aiming to compare the workload experience in two different 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.

Evaluating H1: Iterating Between Type and Expression Construction

- Analyzed data from the Hangman 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."

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.

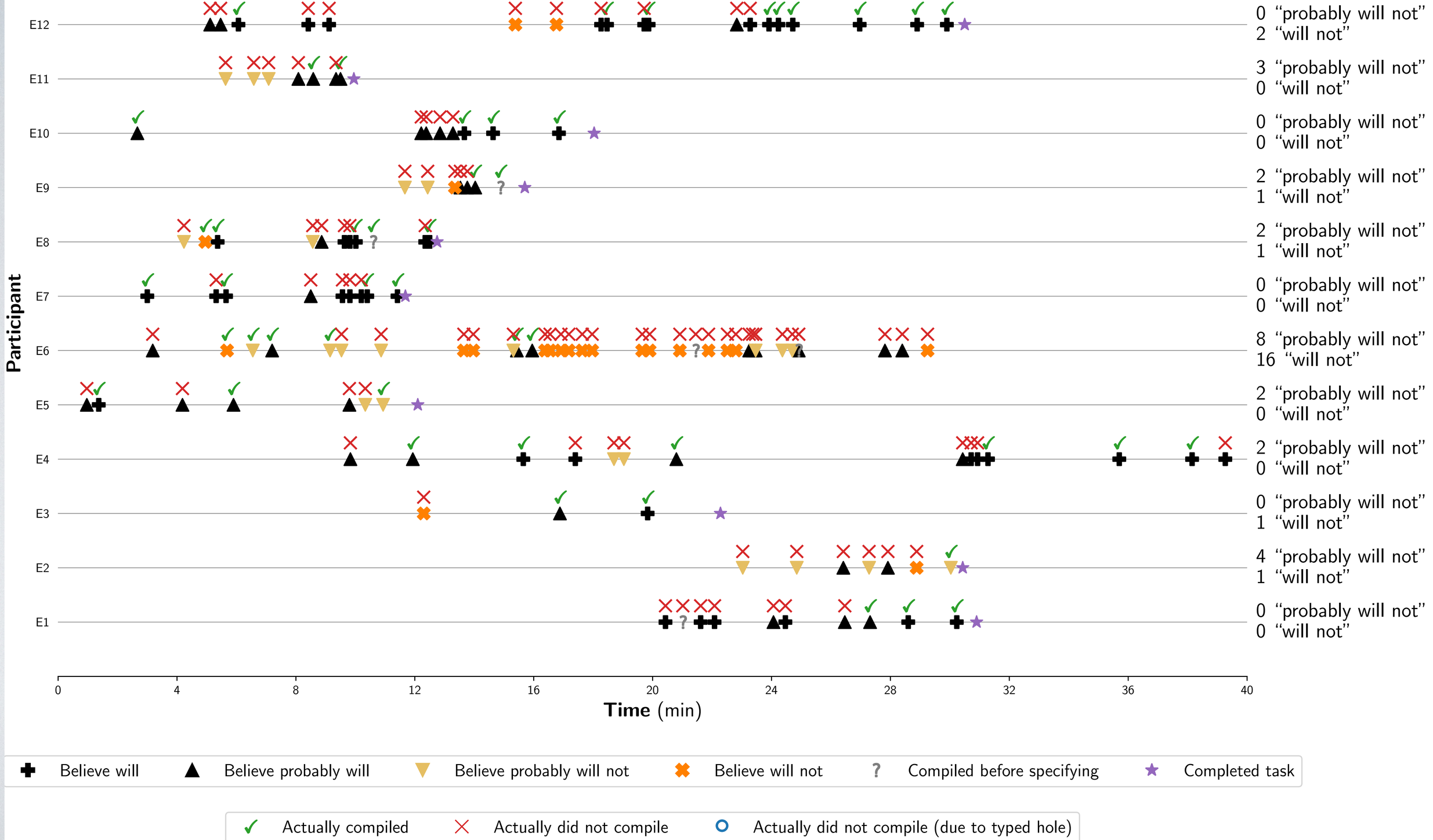


Evaluating H2 (… Run Their Compilers on Code They Know Will Not Compile.)

- 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.

- Classified: 1) creating/modifying type definitions, 2) creating/modifying type signa- tures, 3) creating/modifying expressions, 4) creating/modifying comments, 5) using the REPL, 6) using the internet, or 7) reading the task.


Participant
E12
E11
E10
E9
E8
E7
E6
E5
E4
E3
E2
E1
0 “probably will not”
2 “will not”
3 “probably will not”
0 “will not”
0 “probably will not”
0 “will not”
2 “probably will not”
1 “will not”
2 “probably will not”
1 “will not”
0 “probably will not”
0 “will not”
8 “probably will not”
16 “will not”
2 “probably will not”
0 “will not”
2 “probably will not”
0 “will not”
0 “probably will not”
1 “will not”
4 “probably will not”
1 “will not”
0 “probably will not”
0 “will not”
0
4
8
12
16
20
24
28
32
36
40
Time (min)
Believe will
Believe probably will
Believe probably will not
Believe will not
Compiled before specifying
Completed task
Actually compiled
Actually did not compile
Actually did not compile (due to typed hole)

Implications

- Iteration between type and expression construction => build tools to reduce viscosity of types
- Why does pattern matching reduce workload compared to combinators?