Melody: Function-Oriented Navigation of a Fileless Codebase in Unison

CS279r/CS252r Final Paper

Charlie Colt-Simonds, Matt Neary, Eliza Scharfstein, Ary Swaminathan, and Sreya Vemuri

ABSTRACT

Programmers navigate codebases often and for various reasons: to peer review code, to understand existing code, or to add something new. However, it can be challenging and time-consuming to understand functions with many dependencies across a codebase [1]. We conducted an exploratory study where 7 users navigated a codebase in Unison, a new fileless functional programming language. This study revealed a new challenge for navigating fileless codebases: understanding function dependencies [4] without the artificial organizational structure that files provide. Based on this insight, we designed [5, 9] a new codebase manager for Unison, called *Melody*, to help Unison programmers better understand function dependencies and navigate a codebase. *Melody* displays all function dependencies next to a function and allows users to interactively click through each of these functions, eliminating the difficulties of understanding function dependencies in a fileless codebase. In an evaluative study with 10 participants [8], *Melody* helped programmers successfully debug a Unison function with several layers of dependencies. However, many participants still preferred the user experience of the status quo codebase manager, Unison Codebase Manager (UCM), over *Melody*. From our study results, we present several insights into fileless programming specifically, functional programming more broadly, and integrated development environments in general.

INTRODUCTION

Navigating a new codebase can be difficult, as users run up against a variety of new functions, dependencies, logic, syntax, and styles. Understanding what occurs within a codebase is important for error checking, reviewing, and modifying a segment of code [2]. Programmers encounter new codebases often, as they engage in new projects and peer review other peoples' code. This difficulty of understanding what is occuring in a codebase can be exacerbated when someone is using a new language or a language they use less often, as they are challenged to re-acquaint themselves not only with a particular body of code, but also with the syntax needed to construct it [3].

Most codebases are based on files, and thus afford some automatic—if sometimes arbitrary—organization of functions [1]. Meanwhile, a fileless codebase poses a particular challenge: users still need to be able to navigate through the code and understand function dependencies [4], but the setup lacks the organization that files provide. As such, Unison, a new functional programming language based on this fileless paradigm, is a system that presents a particular need within the broader design space of codebase navigation.

Currently, access to Unison's codebase is primarily through the Unison Codebase Manager (UCM), a command-line accessible tool that handles everything except text editing, including type checking,

executing, browsing of the codebase, refactoring, and publishing. However, the current solution limits the navigability of large codebases [6], and relies heavily on prior knowledge of functions and their role within the codebase. The current solution requires that users know what they want to view and can search for it explicitly. It does not clearly indicate links between functions and it does not permit point-and-click exploration of a code base. Accordingly, the Unison community lacks a codebase manager that adequately addresses users' needs when it comes to navigating and understanding a fileless codebase, as we will discuss later in the paper.

In this paper, we aim to improve upon the Unison codebase manager. In particular, a Unison codebase manager should allow users to access code and minimize onboarding time, load time, and navigation strain—all without resorting to the paradigm of functions grouped into files.

As designers, we aimed to create a solution that alleviates the traditional downfalls of navigating a fileless codebase. In order to better understand the issues that users might run into when trying to understand existing code and write new code in such a fileless system, we ran an exploratory study [8] in which 7 programmers each completed tasks requiring them to understand and modify existing function definitions in a Unison codebase. Observations from this study led to the design of *Melody,* an interactive codebase manager for Unison that provides functionality for visualizing function dependencies and clickable exploration of a Unison codebase. We offer *Melody* as an alternative to the current Unison Codebase Manager.

Melody keeps a dynamically updated dependency graph that it uses to display to users all dependencies for a given function. Each function serves as a hyperlink, giving users the ability to interactively click through functions and navigate through function definitions.

We conducted another controlled study with 10 participants to gain insight into how *Melody* could support fileless codebase navigation in Unison in comparison to UCM. Participants were successful at using *Melody* to debug a function with dependencies across the codebase: 4 out of 5 participants successfully found the error in the provided function in five minutes or less, compared to 0 out of 5 participants with UCM.

Participants enjoyed seeing all the dependencies of a function listed next to it and found the graphical user interface provided by *Melody* intuitive to use, in comparison to UCM. Some participants, however, enjoyed the narrowly scoped view of the codebase that UCM provides and found this view of the codebase easier to work with when focusing in on specific functions and errors. While preference was split amongst participants between UCM and *Melody*, users in both groups expressed a need for codebase managers that aid users in understanding user-generated functions, understanding system-inherent functions, and inferring type signatures of functions. Thus, we suggest a solution that (1) includes a dual-screen display of functions and their dependencies, (2) links functions, including built-in ones to their definitions, to their definitions and dependencies, and (3) displays type definition and function formatting at the top of each function.

In summary, our contributions have both academic and practical applications, which include:

(1) *Melody,* a prototype for a codebase manager for code navigation in Unison, a fileless functional programming language;
(2) two user studies which provide insight into users' usage of *Melody*, UCM, and Unison and functional programming more broadly; and
(3) further recommendations for the design of navigation tools for fileless codebases, functional programming languages, and codebases more generally.

Overview of Unison

Unison is a new type of functional programming language based on two big ideas. First, the programmer's experience of coding need not stay the same it has always been[10]. And second, by a related notion, code should be represented in a content-addressed datastore rather than spread across files [10].

Unison is a language designed around these ideas, and the Unison Codebase Manager (UCM) is the software development toolkit that is packaged with the Unison language [10]. The developer experience in Unison is unique. Developers interact with a codebase using UCM, and to edit the source for existing functions or to define new ones the developer must pull a portion of the codebase out into a scratch file which can then be edited by any ordinary text editor.

We focus on Unison for a variety of reasons: Firstly, our solution has practical benefits for the Unison community in providing an improved codebase manager as described and evidenced in this paper. According to its documentation, Unison as a language as a variety of benefits, including that "it eliminates builds and most dependency conflicts, allows for easy dynamic deployment of code, typed durable storage" [10]. We hope that our contributions will strengthen the usability and usage of Unison so that users can easily access and harness its benefits. Secondly, our user studies have elicited broader insights about functional programming and code base navigation, which we discuss later in this paper.

RELATED WORK

The approach taken in this paper is similar in concept to a number of other concurrent research projects in the field of HCI. While *Melody* is able to display a list of any function's dependencies by leveraging the fileless nature of Unison, others have been able to achieve this functionality in different ways. SourceTrail is an open source graphical code editor that can visualize the structure of a codebase as a graph, but it search through an entire codebase and construct its own model of the dependencies before it can display them visually [13]. Hoogle is a search engine that allows users to search for Haskell libraries by type as well [7, 14]. The closest thing to *Melody* currently is an Elm-based Unison editor created by Paul Chiusano [15]. It is essentially a browser for functions, but is implemented differently than the approach taken here.

DESIGN FICTION

Sam and Riley are unison programmers who co-maintain an API client for the slack API, which many people use to implement slack bots. The API client is written in Unison.

Sam has just updated the codebase and added a function that allows users to specify how the bot will respond to direct messages called `respond`.

Riley wants to review this code before updating the API to ensure that it works properly before pushing it to the working codebase. First, Riley wants to locate the function, but isn't sure what it's called. She opens up the command-line based UCM. She knows there is a command that will list all of the functions, but can't remember what it is, so she goes to her browser and then to the Unison website, scouring a few pages before finding that the command she needs is the `list` command.

She types `list` into the UCM. A large, long list of functions appears, and after reviewing it for a for a minute, Riley finds what she seems to be the correct function based on its naming: `respond`.

```
.> list

  1.  builtin/ (278 definitions)
  2.  l2norm   (Float -> Float -> Float)
  3.  merge    ((a ->{e} a ->{e} Boolean) -> [a] -> [a] ->{e} [a])
  4.  minmax   ([Int] -> (Int, Int))
  5.  mystery  ([a] -> ([a], [a]))
  6.  range    ([Int] -> Int)
  7.  sort     ((a ->{e} a ->{e} Boolean) ->{e} [a] ->{e} [a])
  8.  splitAt  (Nat -> [a] -> ([a], [a]))
  9.  square   (Float -> Float)
  10. uncons   ([a] -> Optional (a, [a]))
```

An example of a response a user would get when typing `list` in the terminal using UCM.

After searching online, Riley reviews the Unison documentation again to find the command that will allow her to view the function. She types `view` into the command line to see the `respond` function. The function type definition and the function itself displays in the terminal.

```
.> view mystery

  mystery : [a] -> ([a], [a])
  mystery s =
    use Nat /
    splitAt (List.size s / 2) s
```

An example of a function accessed through the UCM from the command line. Users see both type definition and the function itself. This function, like `respond`, is dependent on other functions (splitAt) and includes keywords (such as Nat).

Riley appreciates the type definition, but has trouble understanding the function, given that it seems to reference a bunch of other functions (although there is no syntax highlighting, so she has trouble telling what is a function and what is a keyword. She then has to retype each dependent function in command line. She discovers that each of those functions have their own dependencies, and thus has to go through the same `list` command into the command line, and scroll repeatedly back and forth through the dependencies to understand how the `respond` function works. Reading each function is arduous given the UI, and re-typing and scrolling through them take a long time.

Riley finally gets too frustrated with UCM and complains to Sam, who introduces her to *Melody*. Right away, Riley sees that she can search for functions as the top search bar, decreasing the time spent looking through a long list of them, or look through their listed view in the home base of the codebase if she didn't know what a function was called.

Riley locates `response`, clicks on it, and the code for the function and its type definition replaces the home screen. The distinction between functions and keywords is clearly highlighted differentially in the UI,

so Riley has a clue as to what might be happening within it. The details of the dependencies and operators are listed in full in the right-hand window, such that Riley is able to see all the functions at once and does not have to arduously scroll and type back and forth to access them. Riley figures out what the function does, and feels far less frustrated by the process given that the functionality of the various components of the codebase are logically connected in a clear UI.

This fictional user story speaks to the difficulties of navigating a codebase using the existing UCM, which requires the usage of tedious commands, demands prior knowledge of functions, does not distinguish between operators and functions, and warrants a tedious scroll and type back that does not grant users a clear linkage between functions. *Melody* seeks to address these problems through its home function screen, search functionality, dual window display of functions and its dependencies, clear syntax highlighting, and linking of function dependencies within a given functions, allowing for easier and faster navigation and comprehension.

USER STUDIES

We phased our studies into two parts:

(1) an exploratory study to discover where user friction in performing tasks understanding and using code in Unison lie in order to create a prototype for a new Unison codebase manager

(2) a joint evaluative and exploratory study to assess the experience of users using *Melody* in order to generate further design recommendations and insights.

Participants

Participants were recruited from the population of Harvard undergraduates who had completed CS51, an introductory functional programming course. We enrolled a combined total of 17 participants, 2 of whom were female, and 9 of whom were Computer Science concentrators. Participants had a median of 4 years of programming experience, and 1 year of functional programming experience.

User Study 1 - Methods

We conducted an exploratory study in order to understand the process that programmers follow and the obstacles they face when trying to navigate a Unison codebase to understand existing code, modify function definitions, and add new function definitions to the codebase. We observed 7 programmers as they completed two tasks related to understanding a function definition in Unison and modifying the behavior of a specified function. Participants were first given an introduction to the syntax of UCM and Vim, the primary command-line accessible text editor for Unison, such that unfamiliarity with either would not be a factor in evaluating UCM as a codebase manager, and then asked to complete the following two tasks in UCM:

- *Task 1:* modify input parameters to a distributed mergesort, dependent on functions spread out across the Unison codebase
- *Task 2:* explain the behavior of a user-defined function without dependencies spread out across the Unison codebase.

Task 1 could be completed by adding a single input parameter to the (recursive) function and propagating the addition of this parameter to each recursive call within this function. Task 2 could be completed either by observing the results of executing the function with different parameters or by close analysis of the function itself.

After each task, participants reported how difficult they found navigation of the Unison codebase to be using UCM, and suggested features for a Unison codebase manager that would improve their user experience and ability to be successful in completing these tasks. Feedback from participants was split into three categories:

- Qualitative collection of data after each task, regarding what features of UCM aided and inhibited participants in completing the given task
- Comparison of UCM user experience with traditional Interactive Development Environments (IDE) and text editors for both functional and non-functional languages
- Follow-up questionnaire asking participants to quantify (on a scale of 1-5, with 1 being the lowest) usability of UCM and Unison, as well as general comfort with functional programming, imperative programming, and the command line.

User Study 1 - Results

While there is a split in sentiment regarding how easy it is to use Unison, participants agreed across there is room for improvement. On a scale of 1-5, 3/7 rates "Ease of use of Unison:" four, while 4/7 rated it a two or lower. Meanwhile, general comfort with the command line was higher on average than comfort with using the command line in Unison, suggesting this experience could be improved, with 5/7 rating their "comfort with the command line" a four or a five, while 5/7 rated their "comfort with command line in Unison" a three or lower. This discrepancy could be due to the newness of the language; nevertheless, it indicates that the experience for accessing Unison could be improved.

Users also noted specific needs and obstacles in their user interviews. Predominantly, participants called for an improved reference following flow and clearer distinction between code.

<u>Trouble tracing functions and errors</u>

Multiple users expressed difficulty tracing the logic of the functions. In reference to the first task, one participant said: "I tried to look at it and figure out what it did and then my brain exploded...it was hard to figure out what the function was doing just by looking at it." This anecdote captures the difficulty the user faced in understanding a function with a multitude of dependencies.

One user noted the interconnectivity of functional programming, where "every line is connected, so it's harder to trace errors back." Another participant spoke to the difficulty of navigating errors through the command line, where tracing errors back was difficult, saying "When I called merge, it was passing a partially evaluated function...The error wasn't exactly clear to me…" The participant described discovering this error by scrolling up through the code to see the output of the called function. This friction speaks to a need to type check and/or have easy navigation amongst interdependent functions.

Need for syntax highlighting

Many users also called for syntax highlighting within the code to make reading it easier. One participant noted the importance of syntax highlighting, particularly when an individual is unfamiliar with the code or language: “for a new programming language, it’s hard to see what are keywords and what the flow is if there’s no highlighting.” Another said that,“Syntax highlighting and variable name highlighting is good...very much in the bare minimum.” This call for syntax highlighting speaks to the broader desire to make reading code easier, removing the burden of some of the work of comprehension from the user.

Prototype

With insights from the first user study, we built *Melody,* an alternate code base manager to UCM.

Melody brings the basic affordances of hypertext to bear in the navigation of a Unison codebase. Like hypertext, Unison is built on a non-hierarchical web of interconnected text. But in the case of Unison, the text is code and links are a semantic link from a function reference to the function source.

Under the hood, *Melody* interacts with a Unison codebase over the command-line interface. However, the *Melody* backend encapsulates its functionality in an HTTP API that is then called from the frontend. The backend is implemented as a node.js webserver, and the frontend is a web app built with React. The frontend retrieves the set of functions in the codebase from the backend and loads their source code.

With the source code of all functions loaded on the frontend, *Melody* tokenizes the code, identifies the function references, and creates a directed acyclic graph of dependencies across functions. Finally, the user is able to search across functions and click into the details for each of them. A function is presented with its source alongside all its dependencies, enumerated recursively in a topological ordering.

The hope is that this hypertextual interface to Unison codebases will retrieve some of the patterns of code navigation that come more naturally to file-based programming languages, and perhaps point the way forward novel and productive interfaces to codebases more broadly.

User Study 2 - Methods

We conducted a joint evaluative and exploratory study to evaluate the design of *Melody*. We observed 10 programmers as they completed two tasks using either *Melody* or UCM, involving explaining the behavior of a "mystery" function and finding the bug in a specified function. Specifically, participants were asked to:

- *Task 1:* Identify the behavior of the function `mystery`, which splits a list in half
- *Task 2:* Find the error in the function `range`, which should return the range of an integer list

All participants were first given an introduction to *Melody* and then asked to complete Task 1 using *Melody*. Then, participants were randomly selected to either use *Melody* or UCM for Task 2 (those who were tasked with using UCM were then given an introduction to UCM).

Melody *Press ctrl-t to open search, ctrl-q to go home.*

Source for **sort**

```
sort : (a -> a -> Boolean) -> [a] -> [a]
sort lte as =
  if List.size as < 2 then as
  else
    case halve as of
      (left, right) ->
        l = sort lte left
        r = sort lte right
        merge lte l r
```

Dependencies Search full codebase

```
halve : [a] -> ([a], [a])
halve s =
  use Nat /
  splitAt (List.size s / 2) s


merge : (a ->{e} a ->{e} Boolean) -> [a] -> [a] ->{e} [a]
merge lte a b =
  go out a b =
    use List ++
    case (uncons a, uncons b) of
      (None, _) -> out ++ b
      (_, None) -> out ++ a
      (Some (hA, tA), Some (hB, tB)) ->
        if lte hA hB then go (out :+ hA) tA b else go (out :+ hB) a tB
  go [] a b


splitAt : Nat -> [a] -> ([a], [a])
splitAt n as = (List.take n as, List.drop n as)


uncons : [a] -> Optional (a, [a])
uncons as =
  case List.at 0 as of
    None -> None
    Some hd -> Some (hd, List.drop 1 as)
```

An example usage of *Melody*. The user is currently viewing the sort function, and all of its function dependencies are listed on its right. Users can click through and view function dependencies for any of the hyperlinked functions.

Both tasks leveraged the nature of Unison's filelessness and had several layers of dependencies spread out across the codebase. Task 1 was designed as a warm-up exercise to familiarize users with navigating a Unison codebase. Task 2 was designed as an exercise to study user preferences in navigating many layers of function dependencies. Similar to our first user study, we collected several types of data from participants after each task:

- Qualitative feedback regarding what features of *Melody* users thought were helpful in completing the task and what features users would have liked to see in *Melody*
- Qualitative and quantitative comparison of *Melody* user experience with UCM (for those that used both) and other IDEs
- Time taken to complete each task, starting after the explanation of the task, and ending when the participant correctly identifies the function behavior (task 1) or error (task 2)
- Follow-up questionnaire asking participants to quantify (on a scale of 1-5, with 1 being the lowest) usability, ease of navigation, and comprehensibility of functions in *Melody* and UCM, as well as general comfort with the command line, functional programming, and type definitions

User Study 2- Results

While there are no statistically significant results from this study due to small sample

size, and some users indicated preference for UCM over *Melody*, we draw three key insights from the study to help us suggest a modified version of *Melody*: (1) many users appreciated the simultaneous viewing of their functions and the dependencies, (2) users overwhelmingly noted benefiting from type definitions to help comprehend the functions, and (3) users wanted clear reference following for built-in functions to supplement the existing user-defined referenced following.

User Generated Functions Dependency Viewing- Linking and Dual-Screen View

Multiple users mentioned that they appreciated that you could view the function and its dependencies easily through the dual-screen view and linked organization of the functions. One user summed up their satisfaction after clicking on a function by saying,"Oh, wow! That's all the dependencies. That's sick!" The same user praised the dual-screen layout that prevented him from having to scroll "back-and-forth" between the function and its dependency, saying in discussing the second task: "This definitely made my debugging experience a lot easier...if these were just a massive, 400 line sublime text...and I had to figure out what was wrong with range, and min max was at the top and range was at the bottom, scrolling back and forth, I'd probably get pretty frustrated." Another user praised this aspect of *Melody*: "You can quickly go to the relevant functions instead of having to scroll."

Similarly, others users noted more generally that they appreciated the listing of dependencies, with one user stating, "It's helpful, it lists the dependencies of everything," and another commenting on the straightforwardness of the homescreen: "this home screen, where it's all in your face-- I kind of like that. I don't like when they're sorted into three billion categories and you have to figure out the category the function you want is in."

Other users commented on the linking aspect of *Melody*: "Being able to link around and see all of the functions was also super cool;" "I like how you can just click and get all the dependencies, that's pretty cool."

One additional participant summed these findings up by stating: "It's pretty easy to see which things are functions, which things aren't; search is pretty intuitive; clicking on functions to see more detail is intuitive." At least 7/10 of participants commented on the dependency viewing nature of *Melody*. In summary, users appreciated that they could easily see the presence of functions and how functions related to each other through the dual-window display of dependencies and the linking of functions.

Built-In Functions' Definitions

Other users expressed interest in having similar facilities for built-in functions and constructors as for user-generated functions, with one user saying ""It would have been nice to click into built-ins"A variety of users mentioned a difficulty in navigating some of the syntax with one noting that they would have benefited from "Defining what *go* means" and more generally, "definitions of operators, especially if it's a new language." And another mentioning that they "Didn't understand use nat."

One user even suggested having "English descriptions of what the functions do." While we do not think that would necessarily be implementable in *Melody*, this suggestion implies difficulty in understanding the range of functions within *Melody*.

In sum, this evidence supports expanding the existing framework for user-generated function dependency viewing to built-in functions, and implementing a similar support paradigm for constructors.

Type Definition

Listed above each function was the type definition, which is not unique to *Melody* but multiple users' appreciation of its presence is notable. At least 6 users commented on the presence of type definition in *Melody*. Multiple explicitly stated that it was helpful: "Having the type signature was crucial;" "The first thing I noticed was that you have all these type definitions for all your functions here;" "Having the type hint made it intuitive;" "The signature helps a lot." Others described using it in their thought process: noting that "It takes in a list" and outputs "two different lists;" and that they began by "Looking at the header and deciding what the input/output structures are."

These results suggest that any solution must incorporate type definitions prominently, as users found them helpful in deciphering what a function was doing. One user extended this guiding framework to suggest that an "Outline at the top that would show an example layout for a function would be helpful."

While there are no statistically significant results from this study due to small sample size, we highlight three representative user stories from our study to understand the range of experiences in our study:

- User 1 extensively uses the command-line for class projects and primarily uses command-line accessible text editors to program.
- User 2 primarily uses an IDE or text editor application, such as Sublime Text, Atom, or IntelliJ, for programming tasks.
- User 3: User 3 has very little functional programming experience and does not learn new programming languages often, and gets tripped up on the unfamiliar syntax.

While we did not notice convergent trends within a single group, we want to call out that this study included a range of experiences. Participants can be grouped by their speed of task completion as well. Because all users completed task 1 in *Melody*, we compare their speed in the first task to their speed of completion in the second task, when the code base managers used diverged **[Fig. 1].** Participants are sorted by the speed of completion of the first task into *fastest* (0-1 minute); *fast* (1-2 minutes); *slow* (2-3 minutes) and *slowest* (4+ minutes).

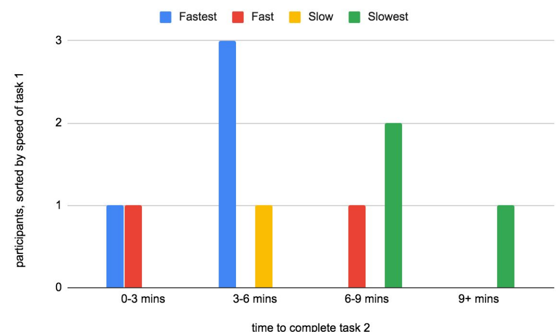

Figure 1. Sorted by their speed in task 1 (fastest being those who completed it in under 1 minute, and slowest being those who took more than 4 minutes to complete the task), we measure the number of participants who took 0-3 minutes, 3-6 minutes, 6-9 minutes, and 9+ minutes to complete task 2.

Figure 1 demonstrates that generally, users who are fast in the first task are fast in the second task.

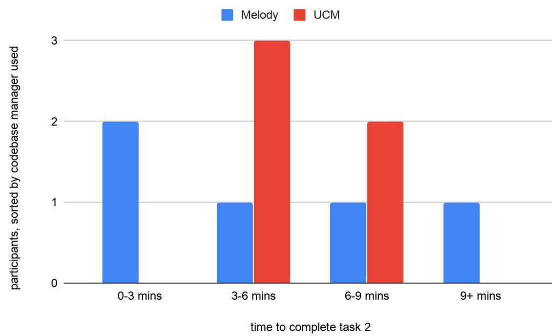

Figure 2. This histogram depicts the distribution of time spent to find the bug and describe it in task 2. In this task, 5 participants completed the task in Melody and 5 in UCM, so the time distribution is delineated by code base manager used.

The fastest completion time occurred for users in *Melody*, although these users also performed within the "fast" or "fastest" groups for the first tasks. Meanwhile, 3 participants within the UCM group performed within the 3-6 minutes group, with ⅔ of them within the fastest group for the first task. These results suggest that time of completion is likely associated with general programming speed, as opposed to usage of UCM v. *Melody*.

Thus, we do not have conclusive trends as to whether *Melody* actually increases the speed of completion for most users, and further study would be required here. Future iterations of this project would have a better control to account for this inconclusivity, as discussed in study limitations. Further, not all users who were exposed to both code managers appreciated *Melody* over UCM, with one noting that in UCM it is "Nice to only see what I need to see."

Thus, *Melody* would benefit from further iteration based on the recommendations at the beginning of the section. In addition, as discussed in the following section, we would hope to test these iterated designs in a more controlled user study in the future in order to more definitively quantify the impact of *Melody* on Unison users.

Study Limitations

Our studies have a few limitations due to small sample size of participants and lack of access to the Unison community that we now discuss. First, although we observed that the fastest task completion times occurred for those using *Melody* rather than UCM, we cannot conclude anything statistically significant about the data because our study is comprised of only 10 participants. Differences in completion times might be related to different levels of programming experience, functional programming experience, and command-line comfort.

Second, due to only having 10 users in our study, we chose to design our study such that all users first used *Melody*. We chose to use this design rather than randomly select between UCM and *Melody* for both tasks in order to have more users overall that used *Melody* so that we could obtain as much feedback as possible. Thus, a limitation of our round 2 user study is that we do not have a true control group. In future studies with more participants, we would have two pairings: (Task 1 - UCM, Task 2 - *Melody*), (Task 1 - *Melody*, Task 2 - UCM) and randomly assign one of these pairs to each participant in order to have an appropriate control group.

Third, participants in both studies had only introductory knowledge of Unison and often had difficulty separating their user experience of understanding Unison syntax with their user experience of using a Unison codebase manager to navigate a Unison codebase. In the intended use case, programmers using this tool will have some

familiarity with the Unison programming language. It might be the case that Unison programmers find *Melody* more useful for navigating a codebase than UCM.

Lastly, the tasks we designed present a limitation in our user studies. Task 2, which required users to either use *Melody* or UCM, had a single error in one dependency. A few users who preferred UCM over *Melody* expressed that UCM was effective for narrowing down the source of an error to a single function. It might be the case, then, that when presented with a task that has interrelated errors across several function dependencies, which is a common occurrence in real programming tasks, users prefer *Melody* over UCM due to its ability to display all these function dependencies at once. In fact, at least one participant who preferred UCM over *Melody* stated that if the given function in Task 2 had been more complex, they might have preferred an alternate codebase manager.

DISCUSSION AND FUTURE WORK

As described, we carried out this study in two phases: (1) an exploratory study to discover where user friction in performing tasks understanding and using code in Unison lie, and (2) a joint evaluative and exploratory study to assess the design and effectiveness of *Melody*, a proposed improvement to UCM. Through these user studies, we gained an understanding of how to better help the Unison community, as well as how to generalize our findings to aid functional programming and codebase management more broadly.

Study Insights

Results from our study show that many of the aspects of *Melody* that users enjoy are not unique to Unison. First, *Melody* has applications in the functional programming space. Many users expressed the difficulty of understanding circular functional dependencies and input or return types in functional programming languages. *Melody* could, thus, be extended to other functional programming languages to allow programmers to better visualize function dependencies and the type signatures of each function. Second, *Melody* has applications to codebase management and interactive development environments, in general. During user studies, several users expressed that they liked not having to scroll around IDEs to search for dependent function definitions or open up several tabs to simultaneously view multiple function dependencies. *Melody* provides a solution to these issues by allowing users to simultaneously view all function dependencies in one window. While *Melody* was initially created for Unison, the concept of creating a dependency graph that is simultaneously viewable is generalizable to functional and non-functional programming. Additionally, below we present other, more general applications of *Melody* to functional programming and codebase management.

Applications to Functional Programming

Functional programming languages have two major language constructs: functions and type constructors. The paradigm employed by *Melody*, of reference-following and displaying function dependencies, can easily be extended to type constructors. Additionally, although Unison does not have modules or classes, the obviously

hierarchical nature of these features suggests the possibility of incorporating their navigation as a feature of *Melody*.

We previously observed the evidence for improving *Melody* to support users that enjoy programming with IDEs and GUIs. Programs written in a functional style lend themselves to various representations such as petri nets or monoidal categories, which in turn can be given a natural graphical syntax as process diagrams or string diagrams. It may be worthwhile to follow up with study participants and obtain their feedback on using graphical tools for interacting with functional code, for example Statebox [11]. Unison is well-suited for extensions to visualization due to focus on purely functional dependencies, and future development of *Melody* could include diagrammatic representations of its dependency views to help users better understand the relationships expressed by functions and types in the codebase.

Application to Codebase Management

The Unison codebase has a non-trivial structural compatibility with git due to its hash tree structure [12]. As a result, it can be used naively in conjunction with version control software while avoiding conflicts. Hence, we expect that any git visualization tools are broadly applicable in their design to the purpose of interacting with the Unison codebase, and additional user studies on these tools could inspire additional functionality for *Melody*, or possibly direct integration into the *Melody* web view.

Limits to Application

Melody would not work well with programming languages where namespace is is dependent on modules or files, or where there is a lack of type signatures. Even in functional languages that appear more likely candidates for adoption, existing codebases might be fundamentally incompatible or impractical to achieve compatibility. Intuitively, functional programming languages that are closer to Unison will be best for *Melody*, but a practical effort toward the goal of bringing compatibility with more functional languages will require a meaningful study and rigorous understanding of the features that are required for a language (or subset of a language) to be sufficiently similar to Unison.

CONCLUSION

We developed *Melody*, an interactive Unison codebase manager to help programmers view function dependencies and navigate a Unison codebase. We recommend a solution that incorporates (1) dual-screen display of functions and their dependencies, (2) linking of functions, including built-in ones to their definitions, and (3) prominently displaying type definition and function formatting at the top of each function. In order to further this work, we would iterate on *Melody,* and perform further user studies with wider sample sizes and tighter controls to further validate the effectivity of the code base manager in this new form.

As it stands, our study shows that programmers can use *Melody* to successfully understand and modify existing codebases. Our results are inconclusive as to whether *Melody* is more effective than the Unison

Codebase Manager (UCM), but we believe that after further iterations, tools like *Melody* have the potential to enable Unison programmers to more quickly and effectively navigate a codebase. Furthermore, we believe that tools like *Melody* may ultimately have the potential to help programmers understand code and its relationships more broadly.

REFERENCES

1. Edward McCormick, Kris De Volder, JQuery: finding your way through tangled code, Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, October 24-28, 2004, Vancouver, BC, CANADA

2. Button, G., Sharrock, W. Occasioned Practices in the Work of Software Engineers. In Jirotka, M., and Goguen, A. (eds.) Requirements Engineering. Social and Technical Issues. Academic Press, London, 1994.

3. Essi Lahtinen , Kirsti Ala-Mutka , Hannu-Matti Järvinen, A study of the difficulties of novice programmers, Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, June 27-29, 2005, Caparica, Portugal

4. Karl J. Ottenstein , Linda M. Ottenstein, The program dependence graph in a software development environment, ACM SIGPLAN Notices, v.19 n.5, p.177-184, May 1984 [doi>10.1145/390011.808263]

5. Jakob Nielsen, Noncommand user interfaces, Communications of the ACM, v.36 n.4, p.83-99, April 1993

6. Nichols, D.M., Thomson, K. & Yeates, S.A. (2001) Usability and open-source software development *Proceedings of the Symposium on Computer Human Interaction*, (eds.) Kemp, E., Phillips, C., Kinshuk & Haynes, J., 49-54. Palmerston North, New Zealand. ACM SIGCHI New Zealand.

7. N. Mitchell, Hoogle overview, The Monad. Reader, 12 (2008), pp. 27-35

8. Andrew Faulring, Brad A. Myers, Yaad Oren, Keren Rotenberg, "A case study of using HCI methods to improve tools for programmers", *Cooperative and Human Aspects of Software Engineering (CHASE) 2012 5th International Workshop on*, pp. 37-39, 2012.

9. Chen Xiajian, Wang Danli, Wang Hongan, "Design and implementation of a graphical programming tool for children", *Computer Science and Automation Engineering (CSAE) 2011 IEEE International Conference on*, vol. 4, pp. 572-576, 2011.

10. "The Unison Language." Accessed December 15, 2019. https://www.unisonweb.org/.

11. Statebox Team: *The Mathematical Specification of the Statebox Language*. Available at http://arxiv.org/abs/1906.07629.

12. Wiegley, John. *Git from the Bottom Up*, jwiegley.github.io/git-from-the-bottom-up/.

13. "Sourcetrail - The Open-Source Cross-Platform Source Explorer." Accessed December 15, 2019. https://www.sourcetrail.com/.

14. "Hoogle." Accessed December 15, 2019. https://hoogle.haskell.org/.

15. "Paul Chiusano: Unison Update 0." Accessed December 15, 2019. https://pchiusano.github.io/2015-01-30/unison-update0.html.