Towards End-user Web Scraping For Customization

Kapaya Katongo
MIT CSAIL
Cambridge, MA, USA
kkatongo@mit.edu

Geoffrey Litt
MIT CSAIL
Cambridge, MA, USA
glitt@mit.edu

Daniel Jackson
MIT CSAIL
Cambridge, MA, USA
dnj@csail.mit.edu

ABSTRACT

Websites are malleable: users can run code in the browser to customize them. However, this malleability is typically only accessible to programmers with knowledge of HTML and Javascript. Previously, we developed a tool called Wildcard which empowers end-users to customize websites through a spreadsheet-like table interface without doing traditional programming. However, there is a limit to end-user agency with Wildcard, because programmers need to first create site-specific adapters mapping website data to the table interface. This means that end-users can only customize a website if a programmer has written an adapter for it, and cannot extend or repair existing adapters.

In this paper, we extend Wildcard with a new system for *end-user web scraping for customization*. It enables end-users to create, extend and repair adapters, by performing concrete demonstrations of how the website user interface maps to a data table. We describe three design principles that guided our system's development and are applicable to other end-user web scraping and customization systems: (a) users should be able to scrape data and use it in a single, unified environment, (b) users should be able to extend and repair the programs that scrape data via demonstration and (c) users should receive live feedback during their demonstrations.

We have successfully used our system to create, extend and repair adapters by demonstration on a variety of websites and we provide example usage scenarios that showcase each of our design principles. Our ultimate goal is to empower end-users to customize websites in the course of their daily use in an intuitive and flexible way, and thus making the web more malleable for all of its users.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

end-user programming, software customization, web scraping, programming by example

ACM Reference Format:
Kapaya Katongo, Geoffrey Litt, and Daniel Jackson. 2021. Towards End-user Web Scraping For Customization. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper is directed at publication. We have provided a PDF but it is best viewed in the online version (kapaya.github.io/px21) which includes videos demonstrating the system.

1 INTRODUCTION

Many websites on the internet do not meet the exact needs of all of their users. End-user web customization systems like Thresher [12], Sifter [13] and Vegemite [19] help users to tweak and adapt websites to fit their unique requirements, ranging from reorganizing or annotating content on the website to automating common tasks. Millions of people also use tools like Greasemonkey [5] and Tampermonkey [7] to install browser userscripts, snippets of Javascript code which customize the behavior of websites.

In our prior work, we presented Wildcard [20], a customization system which enables end-users to customize websites through direct manipulation. It does this by augmenting websites with a table view that shows their underlying structured data. The table is bidirectionally synchronized with the original website, so end-users can easily customize the website by interacting with the table, including sorting and filtering data, adding annotations, and running computations in a spreadsheet formula language. Wildcard enables end-users to be *creators* of browser userscripts (and not just consumers) without having to write Javascript code.

Wildcard has a key limitation. In order to enable end-users to customize a website, a programmer first needs to code a Javascript adapter that specifies how to scrape the website content and set up a bidirectional synchronization with Wildcard's table view. Even though programmers can

share adapters with end-users, this means that an end-user can only use Wildcard on websites where some programmer has already written an adapter. Additionally, if an adapter doesn't scrape the desired data, or stops functioning correctly when a website changes, an end-user has no recourse to extend or repair it on their own.

In this paper, we describe an addition to Wildcard: a system that enables end-users to create, extend and repair website adapters by demonstration within the browser. Using this scraping system, an end-user can perform web customizations using Wildcard on arbitrary websites, without ever needing to code an adapter. Through a series of examples, we show that our system can create Wildcard adapters on a variety of websites via demonstration (Section 2). We also describe key aspects of our system and how web scraping for customization leads to a constraint that simplifies the *wrapper induction* [16] task used to generalize user demonstrations (Section 3).

We then describe the principles underlying the design of our system (Section 4):

- **Unified Environment**: Users should be able to scrape data and interact with the scraped data in a single, unified environment. This minimizes the barrier to fluidly switching back and forth between the two tasks, rather than treating them as entirely independent tasks.
- **Editing By Demonstration**: Users should be able to not only create programs for scraping data by demonstration, but also extend and repair the programs by demonstration. This enables users to build on other users' work, and is especially important in the context of web scraping since scrapers break as the underlying website changes.
- **Live Programming**: Users should receive live feedback as they perform demonstrations. The system should indicate how it is generalizing from the user's example and what the resulting data will look like, so that the user can adjust their demonstrations on the fly and quickly arrive at the desired result.

While each of these principles has been explored in prior work, our contribution in this work is combining them into a novel application to the domain of end user web scraping and customization.

Finally, we share our broader vision for *end-user web scraping for customization*, and some opportunities for future work, including a proposal for how Wildcard's spreadsheet-like formula language might augment demonstrations to provide end-users with more expressiveness in the web scraping process (Section 6).

2 MOTIVATING EXAMPLES

In this section, we show how end-users can create, extend and repair adapters for Wildcard via demonstration. These demos are best viewed as videos in the online version of this paper (https://kapaya.github.io/px21).

2.1 Creating An Adapter

Jen wants to customize her experience on Weather.com by sorting the ten-day forecast based on the description of the weather on each day, allowing her to easily view all the sunny days. She starts the adapter creation process by clicking a context menu item within the Weather.com page, and hovers over a data value she might like to scrape.

The system provides live feedback as Jen hovers, demonstrating the **live programming** principle. The workflow steps are shown in Figure 2:

- The row of data is annotated in the page with a border, to indicate that she will be demonstrating values from within that row (Part 3).
- The column of data is highlighted in the page with a green background, to show how the system has generalized her demonstration across all the rows in the data (Part 4).
- A table view appears at the bottom of the screen, and displays how the values will appear in the data table (Part 5).

Jen tries hovering over several other elements in the page, taking advantage of the live feedback environment to decide what data would be useful. After considering several options, she decides to save the date field in the first column of the table, and commits the action by clicking.

Next, she performs a similar process to fill the next column with the weather descriptions. After filling both columns, she also tries hovering over previously scraped data, and the toolbar at the top of the page indicates which column corresponds to the previously scraped data. Finally, she ends the adapter creation process (Part 7) and is able to immediately sort the forecast by the weather description column, because Wildcard provides a **unified environment** that combines both scraping and customizing.

2.2 Extending An Adapter

Jen has previously used Wildcard to customize timeanddate.com, sorting holidays by day of the week. She comes up with a new customization idea: sorting holidays by category so she can view all the federal holidays together. The current site adapter she is using does not populate the category column in the table, so she needs to extend the adapter. She can immediately perform the extension in the context of the page, using our system's support for **editing by demonstration**.

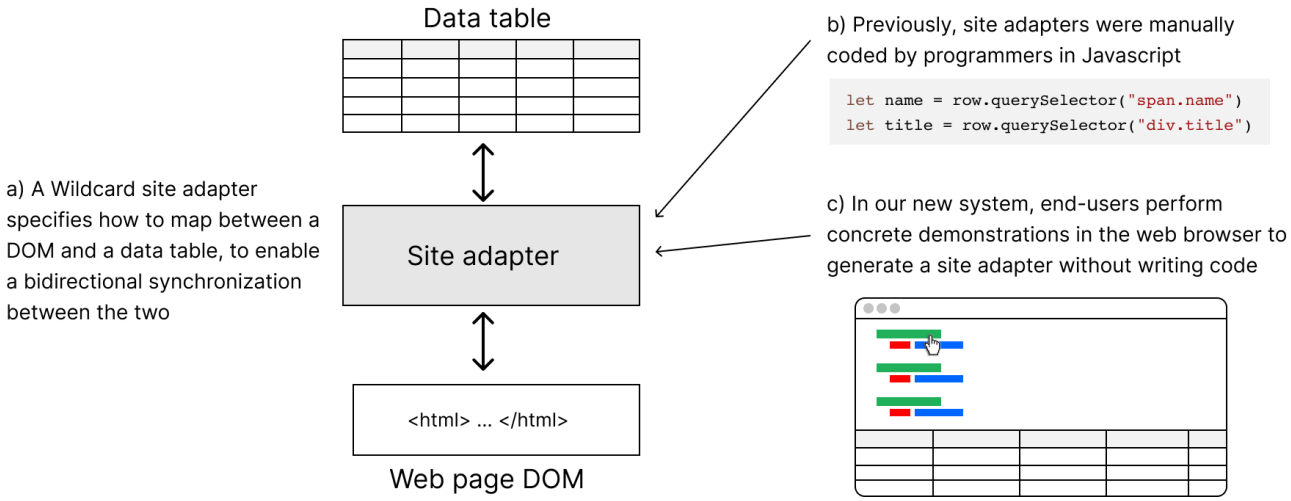

Figure 1: Our work enables end-users to create Wildcard site adapters by demonstration.

The workflow is shown in Figure 3. While viewing the website, she clicks the "Edit Adapter" button (Part 2) above the Wildcard table to initiate the adapter editing process. As she hovers over the currently scraped values, the columns they belong to are highlighted. Finally, she clicks on "Federal Holiday" (Part 3) to add the new column of data to the table (Part 4) and saves the changes (Part 5). Jen then proceeds to sort the list by the type of holiday without the intervention of a programmer.

2.3 Repairing An Adapter

Jen next visits Google Scholar to look up references for a project. Unfortunately, the customization she had applied to sort publications by their title (which is not natively supported by Google Scholar) is no longer working. In fact, the column in the Wildcard table that contained all the publication titles is empty, because the website's internals changed and broke the adapter's scraping logic. Jen can repair this on her own, again taking advantage of **editing by demonstration**.

The workflow is shown in Figure 4. Jen initiates the editing process (Part 2), and initially hovers over the desired value to demonstrate the column she wants to scrape. However, the **live programming** interface indicates to her that the values would be populated into column D; instead, she wants the values to be inserted into column A where they previously appeared. So, Jen clicks on the symbol for column A (Part 3) to indicate that she wants to scrape the values into that column and demonstrates the first publication title (Part 4). The missing values are now back in the table (Part 5). She then proceeds to save her changes (Part 6) and re-apply her customization to the website by sorting the publications by their title.

3 SYSTEM IMPLEMENTATION

We implemented our end-user web scraping system as an addition to the Wildcard browser extension. Prior to this work, website adapters were manually coded in Javascript by programmers. Now, adapters can be automatically created via demonstration. We start by describing our implementations of *wrapper induction* [16], live programming, and editing by demonstration, and then discuss some of the current limitations of our system.

3.1 Wrapper Induction Algorithm

In order to generate reusable scrapers from user demonstrations, our system solves the *wrapper induction* [16] task: generalizing from a small set of user-provided examples to a scraping specification that will work on other parts of the website, and on future versions of the website.

We take an approach similar to that used in other tools like Vegemite [19] and Sifter [13]:

- We generate a single *row selector* for the website: a CSS selector that returns a set of Document Object Model (DOM) elements corresponding to individual rows of the table.
- For each column in the table, we generate a *column selector*, a CSS selector that returns the element containing the column value within that row.

One important difference is that our algorithm only accepts row elements that have direct siblings with a similar structure. We refer to this as the *row-sibling* constraint. Later,

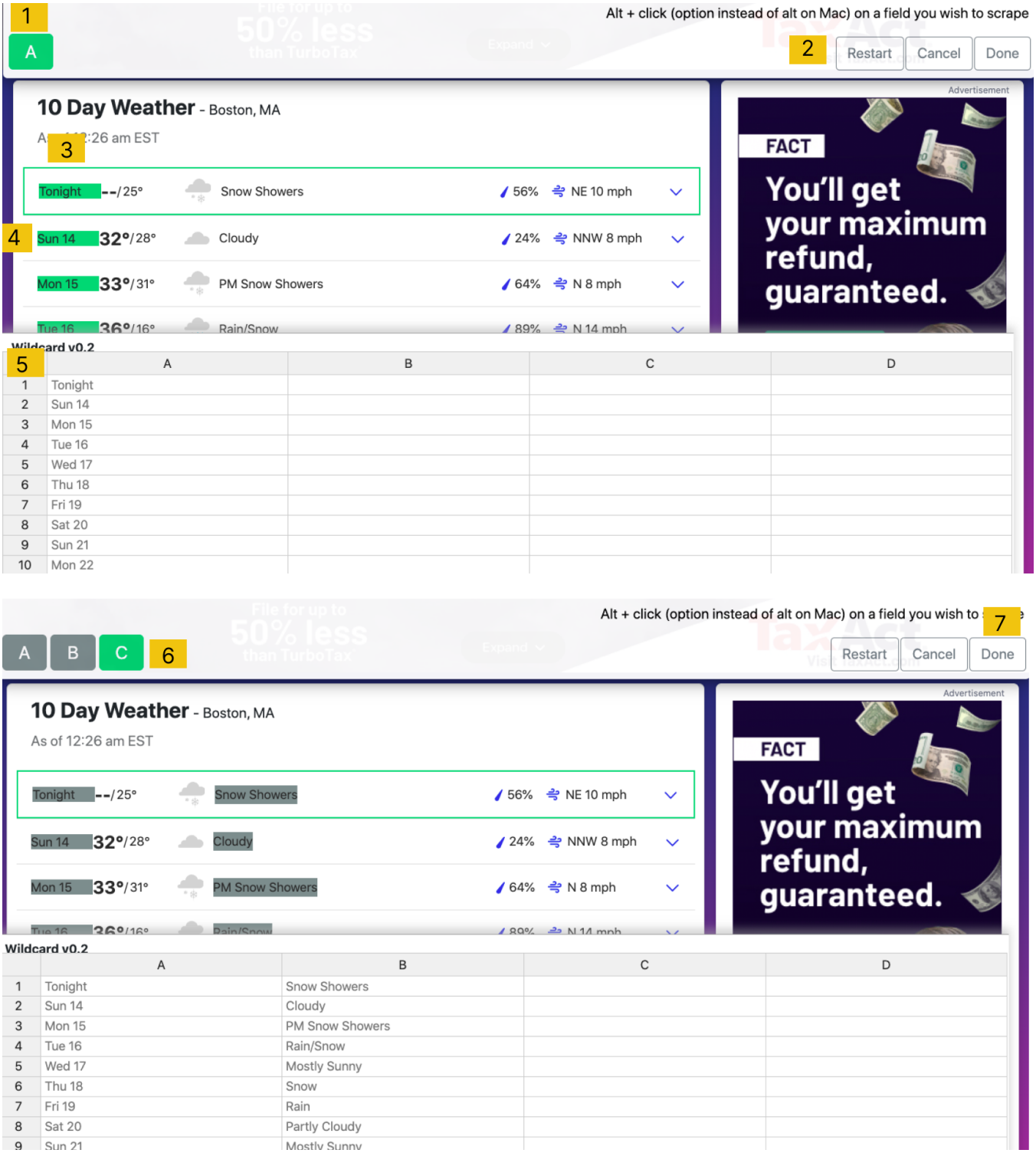

Figure 2: Creating an adapter: 1) column data will be scraped into, 2) system's controls, 3) demonstrated column value and row from which columns should be demonstrated from 4) column values determined by generalization algorithm 5) column values populated in table, 6) next column to scrape data into and 7) button to save adapter created by demonstration

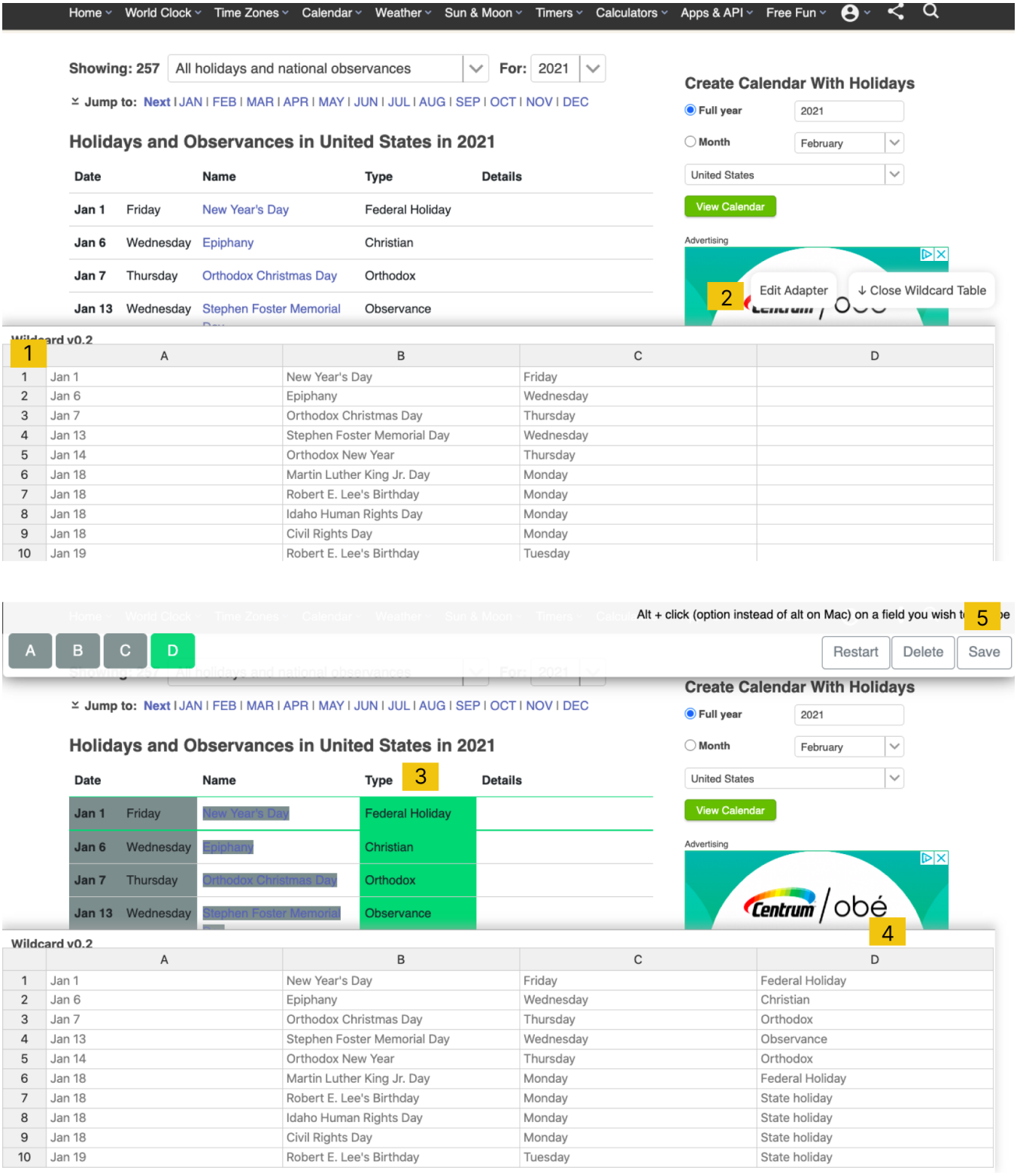

Figure 3: Extending an adapter: 1) table showing current columns and rows, 2) button for initiating adapter extension process, 3) new column value demonstrated and generalized, 4) new column values in the table and 5) button to save adapter extended by demonstration.

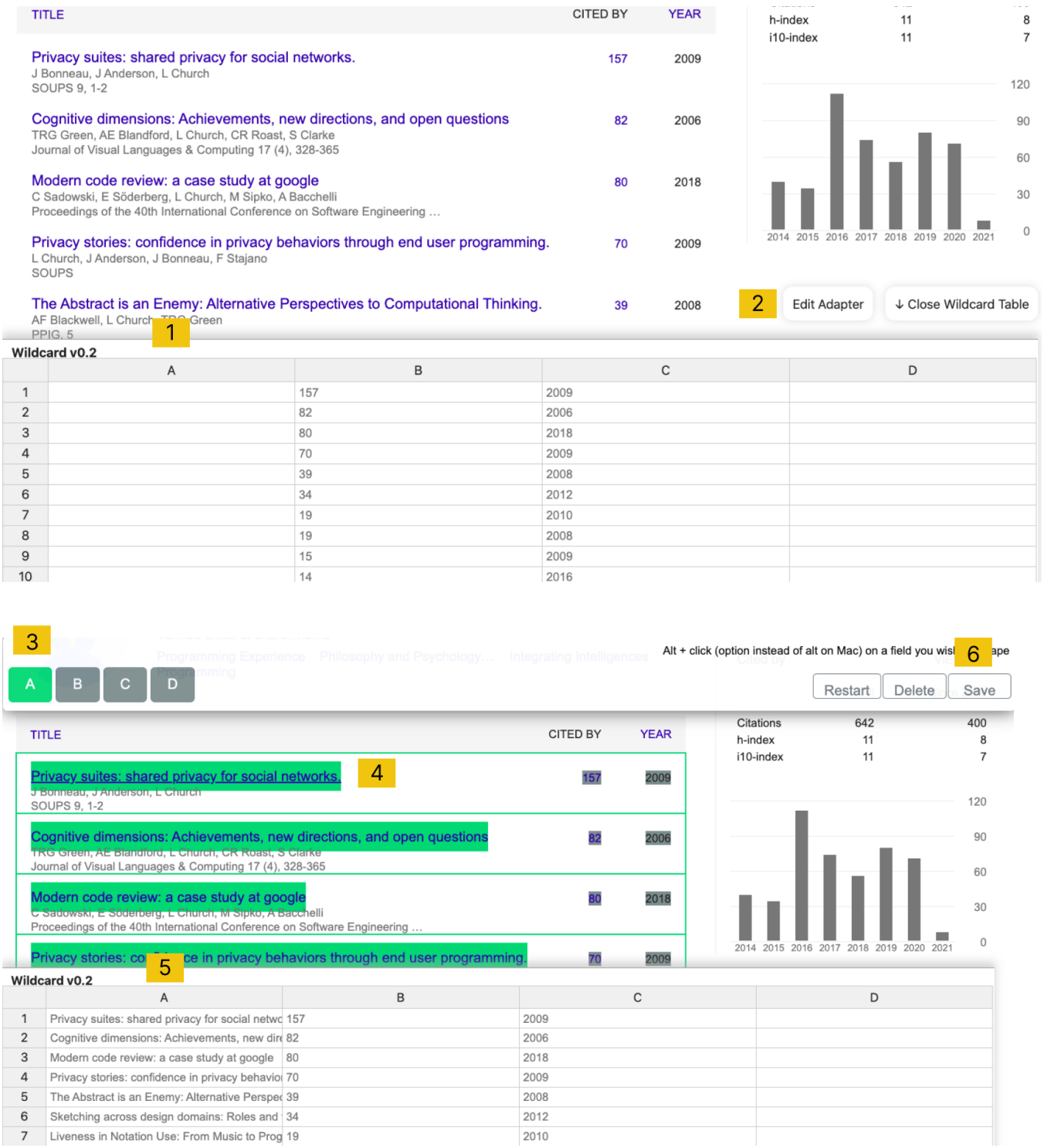

Figure 4: Repairing an adapter: 1) empty column in table because website changed and adapter can no longer scrape values, 2) button for initiating adapter repair 3) column to scrape data into, 4) missing column value demonstrated and generalized, 5) missing column values back in the table and 6) button to save adapter repaired by demonstration.

we describe how the constraint provides a useful simplification of the wrapper induction task and discuss the resulting limitations this puts on our system.

When a user first demonstrates a column value, the generalization algorithm is responsible for turning the demonstration into a row selector that will correctly identify all the row elements in the website and a column selector that will correctly identify the element that contains the column value within a row element. During subsequent demonstrations, the generalization algorithm uses the generated row selector to find the row element that contains the column value and generates a column selector which identifies the corresponding column element.

At a high level, the wrapper induction algorithm's challenge is to traverse far enough up in the DOM tree from the demonstrated element to find the element which corresponds to the row. We solve this using a heuristic; the basic intuition is to find a large set of elements with similar parallel structure. Consider the sample HTML layout in Figure 5, which displays a truncated table of superheroes, with each row containing some nested structure:

The user performs a demonstration by clicking on element *a* in Figure 5 containing "Tony Stark." Our algorithm traverses upwards from the demonstrated element, considering each successive parent element ($b1$, $c1$ and d in Figure 5) as a potential candidate for the row element. For each parent element `el`, the process is as follows:

1. compute a column selector `selector` that, when executed on `el`, only returns the demonstrated element
2. for each sibling `el'` of `el`, execute `selector` on `el'` and record whether the selector returns an element. If it does, this suggests that `el'` has some parallel structure to `el`.
3. compute $el_{siblings}$, the number of sibling elements of `el` which have parallel structure.

Notice how the *row-sibling* constraint simplifies the problem. Row candidates without siblings with parallel structure ($b1$ in Figure 5) have $el_{siblings} = 0$, thus disqualifying them.

The algorithm stops traversing upwards once it reaches the `BODY` element. It chooses the element with the largest positive value of $el_{siblings}$ as the row element, preferring nodes lower in the tree as a tiebreaker. It then generates a *row selector* which returns the row element and all its direct siblings. The final value of `selector` is the column selector since traverses from the row element to the demonstrated data value. These row and column selectors are then used to generate a scraping adapter which returns the DOM elements corresponding to a data row in the table and sets up the bidirectional synchronization.

3.2 Live Programming

The idea of "liveness" in programming can be traced back to Tanimoto's work on VIVA [25]. It generally describes programming environments in which programmers receive immediate feedback about a program while it is being created. In the context of Wildcard, adapters are the program, the table interface is the output of the program and the highlighting on the website is the visual representation of the program. Live programming in our system is implemented by continually re-generating an adapter based on the DOM element under the user's cursor and the previous demonstrations if any, reverting if the user hovers away and committing when the user clicks. The row and column selectors generated during the wrapper induction process are used to highlight all the matching elements on the website and create an adapter. Highlighting all the matching column elements on the website provides visual feedback about the system's generalization to the user. Creating an adapter enables the system to populate the table view and set up the bidirectional synchronization. Because the table is populated and the bidirectional synchronization is set up, users can customize as they scrape. Live programming is possible in our system because the wrapper induction algorithm and adapters execute very quickly. We have yet to benchmark the performance in detail and compare it to other end-user web scraping systems like FlashProg [22] that offer live programming environments.

3.3 Editing By Demonstration

Our system generates adapters with the row selector and the column selectors used to scrape the data. The row selector is a CSS selector that identifies all the row elements of the data and the column selectors are CSS selectors that identify each column's column elements.

When the editing process is initiated, the adapter's row selector and column selectors are used to highlight the previously scraped values on the website. Furthermore, the generalization algorithm takes the adapter's row selector and uses it as the basis to generate new column selectors after each demonstration. When a new column is demonstrated, our system appends the generated column selector to the list of column selectors. This is how adapters are extended to create new columns. When an existing column is demonstrated, our system replaces the column's current column selector with the generated column selector. This is how adapters are repaired to fix broken columns.

Extending and repairing adapters in this manner is feasible because column selectors are independent of each other: changing one column's selector does not affect another column's selector. This is not the case for systems in which the output of demonstrations are dependent on each other.

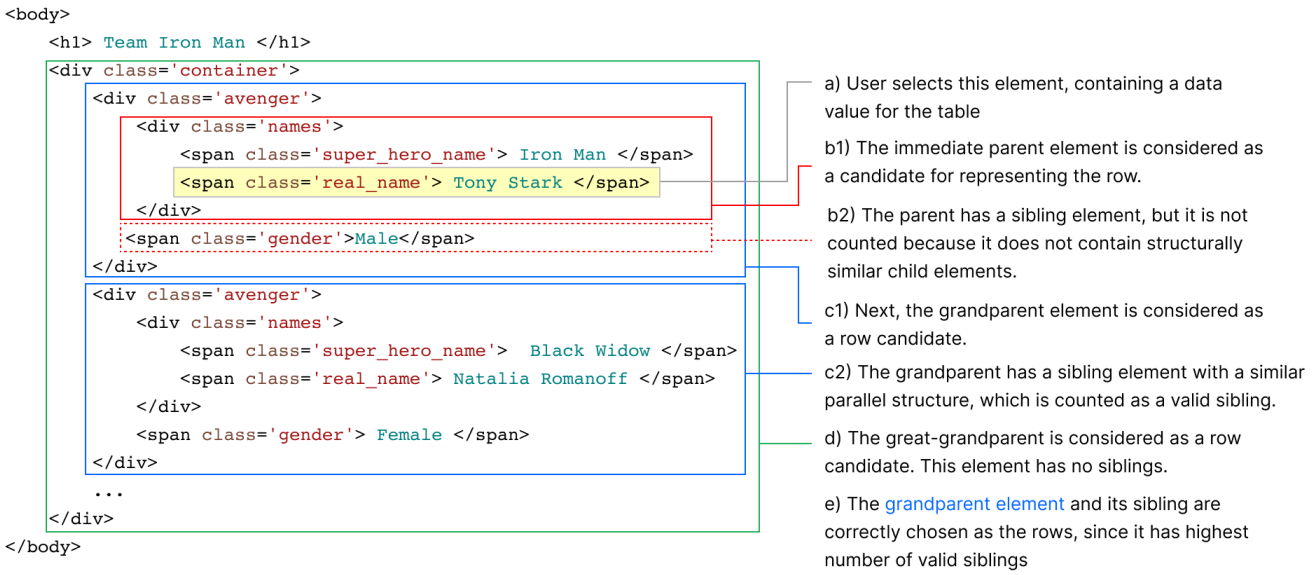

Figure 5: Our system applies a heuristic to identify DOM elements that correspond to rows in the data table.

For example, in a web automation sequence that involves clicking on a button to open a menu and then entering text into the menu's text input, the step that enters the text is not independent because it depends on the step that clicks the button to open the menu.

3.4 Limitations

Since our system is still under development, it has a variety of limitations. In this section we describe two of the most notable ones.

3.4.1 Wrapper Induction Algorithm. The row-sibling constraint we mentioned earlier is important for the end goal of customization because row elements that are not direct siblings may not represent data on the website that should be related as part of the same table by customizations such as sorting and filtering. In Figure 6 we demonstrate two examples where this limitation becomes relevant.

Generalization Limitation 1 shows a case where the data is displayed in a grouped structure. Without the constraint that row elements have to be direct siblings, the row generalization algorithm could determine the row selector to be *.avenger* (elements with blue border) because it matches the largest number of parallel structures (has the largest $el_{siblings}$). While this may be the correct result for the task of extraction, it is not necessarily suitable for the task of customization. When the user sorts and filters, this could result in rows moving between the two tables, disrupting the nested layout and producing a confusing result. Because of this, our system currently does not support such layouts. In the future, we may explore the possibility of extracting multiple tables from a website and joining them together.

Generalization Limitation 2, also in Figure 6, shows a case where the website contains one table of data in which rows are made up of alternating H1 and SPAN tags (elements within blue border). This poses a challenge because each row does not correspond directly to a single DOM element; instead, each row consists of multiple consecutive DOM elements without any grouped structure. Moving the rows when customizing the website would require treating multiple consecutive elements as a single row. This is supported in the underlying Wildcard system, but not yet by our demonstration-based approach.

3.4.2 Data Loaded After Initial Render. Our system currently does not support scraping data loaded after the initial website renders as the user scrolls. Site adapters hand-coded in Javascript can specify event listeners on the DOM to re-execute the scraping code when new data is loaded as a user scrolls. In future work, we plan to provide a mechanism for end-users to specify when a demonstrated adapter should re-execute its scraping code in response to user scrolling. We also do not support scraping data across multiple pages of related data, but this context poses more fundamental challenges to the idea of web customization, since users would somehow need to perform customizations across multiple pages in coordination.

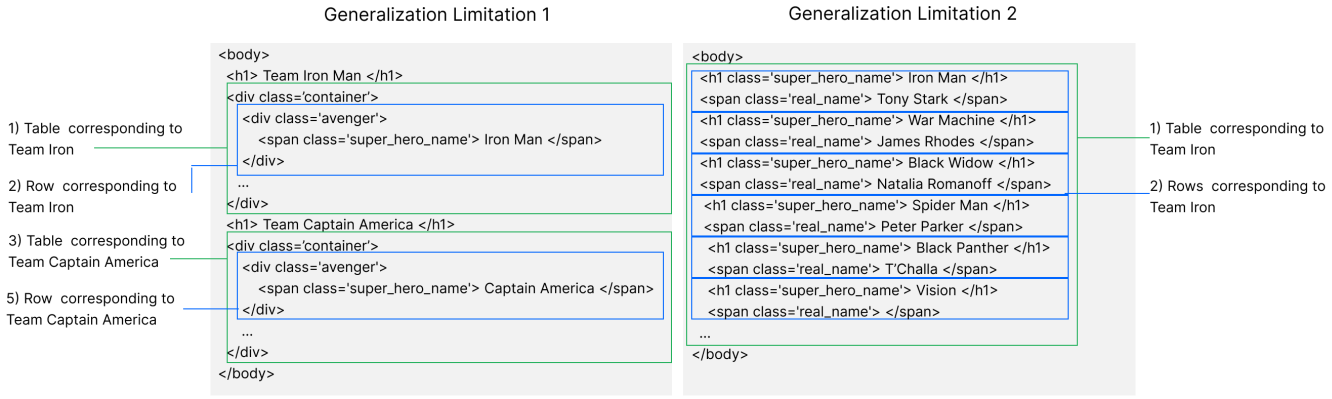

Figure 6: Two example pages where our generalization algorithm does not currently work. The elements with the blue border correspond to rows of the data and the elements with green borders correspond to tables of data in each layout respectively. For the layout on the left, sorting could lead to rows from one table ending up in the other. For the layout on the right, sorting would lead to a distortion of the table since the column elements cannot be moved as a unit.

4 DESIGN PRINCIPLES

Below, we describe our use of three existing design principles in a novel way for the domains of end-user web scraping and customization.

4.1 Unified Environment

In the previous iteration of Wildcard, web scraping was an entirely separate activity from customization. Programmers that wrote scraping adapters would need to switch into an IDE to write code as part of customizing a new website. This divide between tasks is common in other domains:

- In **data science**, workflows revolve between cleaning and using data but this often happens in different environments. The creators of Wrex [11], an end-user programming-by-example system for data wrangling, reported that "although data scientists were aware of and appreciated the productivity benefits of existing data wrangling tools, having to leave their native notebook environment to perform wrangling limited the usefulness of these tools." This was a major reason Wrex was developed as an add-on to Jupyter notebooks, the environment in which data scientists use their data.
- In **web scraping,** if a user comes across an omission while working with data scraped from a website, they need to switch from the environment in which they are using the data to the environment in which they created their scraping code in order to edit and re-run it. This can be seen in many end-user web scraping systems like Rousillon [9]and FlashExtract [17] and commercial tools like import.io [14], dexi.io [10], Octoparse [23] and ParseHub [24].
- In **web customization**, the creators of Vegemite [19], a system for end-user programming of mashups, reported that participants of its user study thought "it was confusing to use one technique to create the initial table, and another technique to add information to a new column." This hints at the need for both a unified environment and a unified workflow.

In this work, we have combined scraping and customization into a single, unified environment with a unified workflow. The goal is to minimize the environment switch between *extracting* the data and *using* the data. A user might start out by scraping some data on a website, and then switch to customizing the website using the results. Then, they might realize they need more data to perform their desired task, at which point they can easily extend the adapter by demonstrating new columns. All of these tasks take place right in the browser, where the user was initially already using the website. Instead of bringing the data to another tool, we have brought a tool to the data. This principle relates to the idea of "in-place toolchains" [4] for end-user programming systems: users should be able to program using familiar tools in the context where they already use their software.

Of course, there is value in specialized tools: Wildcard has nowhere near the full capabilities of spreadsheet software or

databases. Nevertheless, we believe a single, unified environment for scraping and customization presents a significantly lower barrier to entry for customization.

4.2 Editing By Demonstration

Many end-user web scraping and macro systems allow users to *create* programs by demonstration but do not offer a way to *edit* them by demonstration. In Rousillon [9], a web scraping program created by demonstration can only be edited through a high-level, block-based programming language called Helena [6]. Helena supports adding control flow logic (conditional execution, wait times etc) which is invaluable for automating access to websites. However, it does not support extending the web scraping code to add new columns after the demonstration or repairing it to provide new selectors if the website changes. In Vegemite [19], a web automation program created through demonstration can only be edited by editing the text-based representation of the automation demonstrations. In fact, only the demonstrations used to perform automations on the scraped website data can be edited. If a user needs to add a new column or repair an existing one in the scraped data table, they need to re-demonstrate the columns and then re-run the automation script. One exception to existing editing models worth pointing out is import.io [14]. It allows users to add new columns by demonstration but it is not clear whether deleting a column and re-demonstrating it could serve the purpose of repair.

In the prior iteration of Wildcard, if a website's hand-coded adapter stops working correctly because the website changes, an end-user's customizations will often break too. Furthermore, end-users cannot extend the scraping adapter to add columns to the table in order to perform new customizations. This goes against MacLean et. al.'s vision of user-tailorable systems [21] that give users "a feeling of ownership of the system, to feel in control of changing the system and to understand what can be changed." Providing an easy way for users to edit programs is therefore fundamental to fully democratizing web customization.

Editing by demonstration makes end-users first-class citizens in the customization ecosystem. Because users interact with the scraped data through a unified environment directly in the context of the website, it is easy to initiate the scraping system in editing mode: the scraping system is simply booted up using metadata stored with the scraping adapter to the state when the demonstration was completed. Users that have gone through the creation process will immediately realize what to do in order to extend or repair the adapter. Users that have not gone through the creation process might have a harder time but we provide visual clues (such as highlighting the row to perform demonstrations from with a green border) and live programming (immediately preview the results of demonstrations) that serve as guides.

As discussed in Section 3, editing by demonstration in the web scraping domain is feasible because column selectors are independent of each other. However, this is not the case with row selectors because column selectors are dependent on them. Our system therefore does not support editing rows but this an acceptable limitation given our focus on extension and repair which only involve column selectors.

4.3 Live Programming

In many programming-by-demonstration web scraping systems [9, 19], users only get *full* feedback about the program's execution (result of wrapper induction and the scraped values) after providing *all* the demonstrations. This means they cannot adjust their demonstrations in response to the system's feedback as they demonstrate.

Our end-user web scraping system provides level 3 liveness under Tanimoto's liveness hierarchy [25]. Level 3 liveness describes a system that is constantly listening for actions from the user, automatically re-constructing and re-executing the program whenever one happens instead of requiring an explicit command from the user. To eliminate the described edit-compile-debug cycle, our system automatically runs the wrapper induction algorithm and generates an adapter after each user demonstration. As we showed in Section 2, when a user demonstrates a value of a column they wish to scrape, our system immediately shows how it has generalized the user's demonstration across the other rows of the data by highlighting the all relevant values on the website. It also populates the table with the scraped data based on the latest demonstration. The highlighting and table population serve to provide a visual representation of the adapter's execution.

Many successful end-user programming systems such as spreadsheets and SQL provide users with immediate results after entering commands. Our live programming environment is particularly similar to that of FlashProg [22], a framework that provides user interface support for programming-by-demonstration systems like FlashExtract [17], and relates to the idea that an important quality of end-user programming is "interaction with a living system" [4].

Unlike text-based commands which are only valid once complete (e.g `SELECT * FRO` versus `SELECT * FROM user_table`), the target of demonstration commands (the value of a DOM element under the cursor) is the same during both hover and click. This allows us to execute a command before a user completes it, thereby providing them with a preview of the results on hover.

There are limits to this approach. Providing live feedback on websites with a large number of DOM elements or complex CSS selectors can slow down the generalization process, especially if a user is constantly moving their cursor. Furthermore, many datasets are too large to preview in the table in their entirety; the user might benefit more from the live feedback if it could summarize large datasets. For example, FlashProg provides a summary of the generalization through a color-coded minimap next to the scrollbar of its extraction interface.

5 RELATED WORK

End-user web scraping for customization relates to existing work in end-user web scraping and end-user web customization by a number of tools.

5.1 End-user Web Scraping

FlashProg [22] is a framework that provides user interface support for FlashExtract [17], a framework for data scraping by examples. FlashProg's interface provides immediate visual feedback about the generalization and scrapes the matched values into an output tab. In addition, it has a program viewer tab that contains a high level description of what the generated program is doing and provides a list of alternative programs. Finally, it has a disambiguation tab that utilizes conversational clarification to disambiguate programs, the conservations with the user serving as inputs to generate better programs. Though FlashProg has many desirable features we aim to implement in future iterations, its implementation does not align with our goal to provide a unified environment within a browser for scraping and customizing websites.

Rousillon [9] is a tool that enables end-users to scrape distributed, hierarchical web data. Because demonstrations can span across several websites and involve complex data access automation tasks, its interface does not provide *full* live feedback about its generalizations or the values to be scraped until all the demonstrations have been provided and the generated program has been run. If run on a website it has encountered before, Rousillon makes all the previously determined generalizations visible to the user by color-coding the values on the website that belong to the same column. This is a desirable feature for our system as users will not have to actively explore in order to discover which values are available for scraping and how they are related to each other. On the extension and repair front, Rousillon presents the web scraping code generated by demonstration as an editable, high-level, block-based language called Helena [6]. While Helena can be used to perform more complex editing tasks like adding control flow, it does not support adding or repairing columns after the demonstrations and presents a change in the model used for creation. Our system maintains the model used for creation by allowing users to extend and repair web scraping code via demonstration.

5.2 End-user Web Customization

Vegemite [19] is a tool for end-user programming of mashups. It has two interfaces: one for scraping values from a website and another for creating scripts that operate on the scraped values. The web scraping interface does not provide live feedback about the generalization on hover but after a user clicks a value, the interface shows the result of the system's generalization by highlighting the all matched values. Furthermore, even though the interface also has a table, the table is only populated with the scraped values after all the demonstrations have been provided. The scripting interface utilizes CoScripter [18] which is used to record operations on the scraped values for automation. For example, the scripting interface can be used to demonstrate the task of copying an address in the data table, pasting it into a walk score calculator and pasting the result back into the table. The script would then be generalized to all the rows and re-run to fill in the remaining walk scores. CoScripter provides the generated automation program as text-based commands, such as "paste address into 'Walk Score' input," which can be edited after the program is created via "sloppy programming" [19] techniques. However, this editing does not extend to the web scraping interface used for demonstrations and presents a change in the model used for creation.

Sifter [13] is a tool that augments websites with advanced sorting and filtering functionality. Similarly to Wildcard, it uses web scraping techniques to extract data from websites in order to enable customizations. However, Wildcard supports a broader range of customizations beyond sorting and filtering, including adding annotations to websites and running computations with a spreadsheet formula language. Our scraping system intentionally provides less automation than Sifter. Sifter attempts to automatically detect items and fields on the page with a variety of clever heuristics, including automatically detecting link tags and considering the size of elements on the page. It then gives the user the option of correcting the result if the heuristics do not work properly. In contrast, our heuristics are simpler and make fewer assumptions about the structure of websites. Rather, we give more control to the user from the beginning of the process, and incorporate live feedback to help the user provide useful demonstrations. We hypothesize that focusing on a tight feedback loop rather than automation may support a scraping process that is just as fast as an automated one, but gives the user finer control and extends to a greater variety of websites where more complex heuristics do not apply.

However, further user testing is required to actually validate this hypothesis.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented our progress towards *end-user web scraping for customization*, to empower end-users in Wildcard's ecosystem to create, extend and repair scraping adapters. There are several outstanding issues and open questions we hope to address in future work.

Like existing programming-by-demonstration approaches, web scraping in our current implementation is limited to what can be demonstrated by point-and-click. More generally, this surfaces a fundamental limitation of programming-by-demonstration: the inability to specify logic. One solution for this is taking advantage of spreadsheet formulas which have enabled millions of end-users to specify logic. Because of this, Wildcard already includes a spreadsheet-like formula language for specifying customizations. We plan to extend this language to augment our demonstration-based web scraping with the goal of raising the ceiling on the expressiveness available to end-users. One concrete use case for this is scraping DOM element attributes which cannot be demonstrated by point-and-click but contain valuable data. For example, link elements have an *href* attribute which contain the URLs associated with the link element and video elements have *currentTime* and *duration* attributes which contain data about the current playback position and duration respectively. This use of spreadsheet-like formulas to specify logic is related to approaches taken by Microsoft Power Apps [2], Glide [1], Coda [3] and Gneiss [8].

To assess our design principles, we plan to carry out a broader evaluation of our system through a user study. So far, we have only tested the system amongst ourselves and a small number of colleagues. More testing is needed to understand whether it can be successfully used among a broader set of users across a wider variety of websites. We also plan to provide more insight, feedback and control into the wrapper induction process. This is particularly important when the system produces an error or is not able to generate an adapter for a website. One solution for this would be to incorporate a version of the program viewer and disambiguation features of FlashProg [22]. Wrangler [15], whose interface is also centered around a table representation, could also provide some inspiration.

Our ultimate goal is to empower end-users to customize websites in the course of their daily use in an intuitive and flexible way, and thus make the web more malleable for all of its users.

REFERENCES

[1] [n.d.]. *Build an App from a Google Sheet in Five Minutes, for Free • Glide.* https://www.glideapps.com/

[2] [n.d.]. *Business Apps | Microsoft Power Apps.* https://powerapps.microsoft.com/en-us/

[3] Coda | A new doc for teams. [n.d.]. *Coda | A New Doc for Teams.* Coda | A new doc for teams. https://coda.io/welcome

[4] [n.d.]. *End-User Programming.* https://www.inkandswitch.com/end-user-programming.html

[5] [n.d.]. *Greasemonkey - GreaseSpot Wiki.* https://wiki.greasespot.net/Greasemonkey

[6] [n.d.]. *Helena | Web Automation for End Users.* http://helena-lang.org/

[7] [n.d.]. *Tampermonkey for Chrome.* http://www.tampermonkey.net

[8] Kerry Shih-Ping Chang and Brad A. Myers. [n.d.]. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu Hawaii USA, 2014-10-05). ACM, 87–96. https://doi.org/10.1145/2642918.2647371

[9] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. [n.d.]. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology - UIST '18* (Berlin, Germany, 2018). ACM Press, 963–975. https://doi.org/10.1145/3242587.3242661

[10] dexi.io. [n.d.]. *The Most Powerful Web Scraping Software Available.* https://webscraping.dexi.io

[11] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. [n.d.]. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu HI USA, 2020-04-21). ACM, 1–12. https://doi.org/10.1145/3313831.3376442

[12] Andrew Hogue and David Karger. [n.d.]. Thresher: Automating the Unwrapping of Semantic Content from the World Wide Web. In *Proceedings of the 14th International Conference on World Wide Web - WWW '05* (Chiba, Japan, 2005). ACM Press, 86. https://doi.org/10.1145/1060745.1060762

[13] David F. Huynh, Robert C. Miller, and David R. Karger. [n.d.]. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST '06* (Montreux, Switzerland, 2006). ACM Press, 125. https://doi.org/10.1145/1166253.1166274

[14] import.io. [n.d.]. *Data Scraping | Web Scraping | Screen Scraping | Extract.* Import.io. https://www.import.io/product/extract/

[15] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. [n.d.]. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems - CHI '11* (Vancouver, BC, Canada, 2011). ACM Press, 3363. https://doi.org/10.1145/1978942.1979444

[16] Nicholas Kushmerick. [n.d.]. Wrapper Induction: Efficiency and Expressiveness. 118, 1 ([n. d.]), 15–68. https://doi.org/10.1016/S0004-3702(99)00100-9

[17] Vu Le and Sumit Gulwani. [n.d.]. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh United Kingdom, 2014-06-09). ACM, 542–553. https://doi.org/10.1145/2594291.2594333

[18] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. [n.d.]. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2008-04-06) *(CHI '08)*. Association for Computing Machinery, 1719–1728. https://doi.org/10.1145/1357054.1357323

[19] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. [n.d.]. End-User Programming of Mashups with Vegemite. In

Proceedings of the 14th International Conference on Intelligent User Interfaces (New York, NY, USA, 2009-02-08) *(IUI '09)*. Association for Computing Machinery, 97–106. https://doi.org/10.1145/1502650.1502667

[20] Geoffrey Litt and Daniel Jackson. [n.d.]. Wildcard: Spreadsheet-Driven Customization of Web Applications. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming* (Porto, Portugal., 2020). Association for Computing Machinery, 10. https://doi.org/10.1145/3397537.3397541

[21] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. [n.d.]. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1990-03-01) *(CHI '90)*. Association for Computing Machinery, 175–182. https://doi.org/10.1145/97243.97271

[22] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. [n.d.]. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte NC USA, 2015-11-05). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

[23] Octoparse. [n.d.]. *Web Scraping Tool & Free Web Crawlers | Octoparse*. https://www.octoparse.com/#

[24] ParseHub. [n.d.]. *ParseHub | Free Web Scraping - The Most Powerful Web Scraper*. https://www.parsehub.com/

[25] Steven L. Tanimoto. [n.d.]. VIVA: A Visual Language for Image Processing. 1, 2 ([n. d.]), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6