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 install browser extensions and run arbitrary Javascript in the developer console to change them. However, this malleability is only accessible to programmers with knowledge of HTML, CSS, Javascript and the DOM. To broaden access to customization, our prior work developed an approach that empowers end-users to customize websites without traditional programming via a browser extension called Wildcard. Wildcard's customizations are powered by web scraping adapters which are currently written in Javascript by programmers. This means that end-users can only customize a website if a programmer has written an adapter for it. Furthermore, end-users do not have the ability to extend adapters in order to perform new customizations or repair adapters to fix broken customizations.

In this paper, we extend Wildcard with a new system for *end-user web scraping for customization* which enables end-users to create, extend and repair adapters by demonstration. 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 program that scrapes data via demonstration and (c) users should receive live feedback when providing demonstrations to scrape data.

We have successfully used our system to create, extend and repair adapters by demonstration on a variety of websites and provide example usage scenarios that showcase each of our design principles.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
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 browser extensions

ACM Reference Format:
Kapaya Katongo, Geoffrey Litt, and Daniel Jackson. 2021. End-user Web Scraping For Customization. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many websites on the internet do not meet the exact needs of all of their users. End-user web customization systems like Chickenfoot [3], Thresher [6], Sifter [7] and Vegemite [11] help users tweak and adapt websites to fit their unique requirements, ranging from reorganizing or annotating content on the website to automating common tasks. In our prior work, we presented Wildcard [12], 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 has a key limitation. In order to enable end-users to customize a website, a Javascript programmer first needs to code an adapter that specifies how to scrape the website content and set up a bidirectional synchronization with Wildcard's table view. Take Alice, an end-user who uses Wildcard to customize her experience on Google Scholar as shown above. Wildcard's sorting customization gives her the power to sort publications by title which the website does not natively allow. She thinks this will be useful on Weather.com to sort the ten day forecast by the weather descriptions so that she can quickly find all the sunny days. Unfortunately, a programmer has not coded an adapter for Weather.com so Alice is unable to customize it. Additionally, if the adapter for Google Scholar stops functioning when the

website changes, Alice has no recourse to repair it on her 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 end-to-end web customizations using Wildcard on arbitrary websites, without ever needing to code an adapter. Through a series of examples, we show that end-users can utilize our system to successfully 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 reveals a constraint that simplifies the *wrapper induction* [8] task used to generalize user demonstrations (Section 3).

Our key contribution is a set of three design principles that guided the development of our system, which also offer insights that might be applied to other end-user web scraping and customization tools (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 user's work, and is especially important in the context of web scraping since programs that scrape data sometimes 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.

Finally, we share our broader vision for 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.

2.1 Creating An Adapter

Alice, our end-user from Section 1, 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 Alice hovers, demonstrating the **live programming** principle:

- The selected row of data is annotated in the page with a border, to indicate that she will be demonstrating values from within that row.
- The selected 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.
- A table view appears at the bottom of the screen, and displays how the values will appear in the data table.

Alice 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 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

Alice has previously used Wildcard to customize timeanddate.com. In addition to sorting the list of holidays in a year by the day of the week, she also wants to sort them by category so she can view all the federal holidays together. Previously, she would have needed to find a programmer to help her edit the adapter code to incorporate this additional data, but using our system's support for **editing by demonstration**, Alice can extend the adapter herself.

While viewing the website, she clicks the "Edit Adapter" button 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" to add the new column of data and saves the changes. Alice then proceeds to sort the list by the type of holiday without the intervention of a programmer.

2.3 Repairing An Adapter

Alice next visits Google Scholar to look up references for her thesis project. Unfortunately, the customization she had

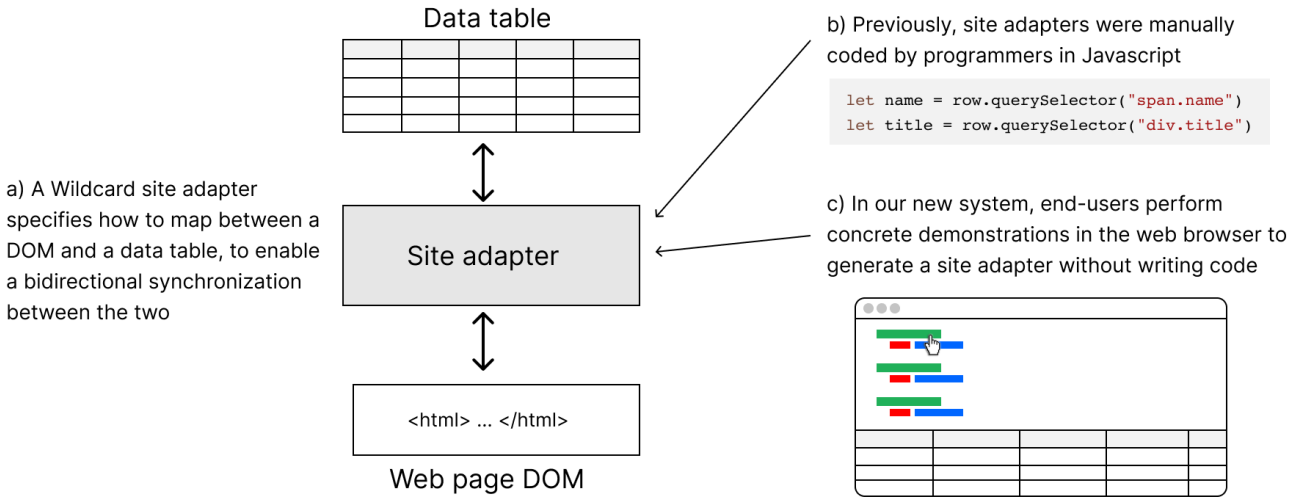

Figure 1: Our work enables end-users to create Wildcard site adapters by demonstration.

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. Alice can repair this on her own, again taking advantage of **editing by demonstration**.

Alice initiates the editing process, 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, Alice clicks on the symbol for column A to indicate that she wants to scrape the values into that column. She then proceeds to re-apply her customization to the website by sorting the publications by their title without the intervention of a programmer.

3 SYSTEM IMPLEMENTATION

We implemented our end-user web scraping system as an addition to the Wildcard browser extension. We start by describing our implementations of row and column generalization, live programming, and editing by demonstration, and then discuss some of the current limitations of our system.

3.1 Generalization Algorithms

In order to generate reusable scrapers from user demonstrations, our system solves the *wrapper induction* [8] 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 [11] and Sifter [7]:

- We generate a single *row selector* for the website: a CSS selector that returns a set of 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 in this section, we describe how the constraint provides a useful simplification of the wrapper induction task and discuss the resulting limitations this puts on our system in a section that follows.

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 generalization 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 following sample HTML layout, which displays

a truncated table of superheroes, with each row containing some nested structure:

```
<body>
    <h1> Team Iron Man </h1>
    <div class='container'>
        <div class='avenger'>
            <div class='names'>
                <span class='super_hero_name'> Iron Man </span>
                <span class='real_name'> Tony Stark </span>
            </div>
           <span class='gender'> Male </span>
        </div>
        <div class='avenger'>
            <div class='names'>
                <span class='super_hero_name'>  Black Widow </span>
                <span class='real_name'> Natalia Romanoff </span>
            </div>
            <span class='gender'> Female </span>
        </div>
        ...
    </div>
</body>
```

The user performs a demonstration by clicking on the SPAN element containing "Tony Stark." Our algorithm traverses upwards from the demonstrated element, considering each successive parent element as a potential candidate for the row element. For each parent element n, the process is as follows:

1. compute a selector s that when executed on n only returns the demonstrated element
2. for each sibling m of n, execute s on m and record whether the selector returned an element. Intuitively, if the selector returns an element, this suggests that the sibling m has some parallel structure to n
3. compute $n_{siblings}$, the number of sibling elements of n for which s returned an element

Notice how the row-sibling constraint simplifies the problem: row candidates without siblings that return an element after s is executed on them have $n_{siblings} = 0$, thus disqualifying them. Furthermore, a candidate element n is only accepted as the row element if the row selector associated with it only matches itself and elements that are its direct siblings.

From the parent of the initial SPAN element, the algorithm moves up the DOM tree until it reaches the BODY element. Then, it generates a selector for each n and discards any n with a selector that matches elements that are either not itself or a direct sibling. Finally, the algorithm picks an n with the largest, positive $n_{siblings}$, preferring nodes lower in the tree as a tiebreaker. s is used as the selector for the column.

Applying this to our example, if the user clicks on the SPAN element containing "Tony Stark," , the first parent element is the DIV with the class *names*This DIV has a sibling SPAN element, but the sibling doesn't return an element when *.super_hero*, the selector that identifies the initial SPAN in the DIV, is executed on it. As a result, it has $n_{siblings} = 0$. At the level above that, we consider the DIV with class *avenger*, which has at least one sibling that returns an element when the selector *.super_hero* is executed on it. Finally, the level above that once again has no siblings that return an element when the selector *.super_hero* is executed on them. Therefore, our algorithm returns the DIV element and outputs *.avenger* as the row selector and *.super_hero* as the column selector. These selectors are used to generate a DOM scraping adapter which returns the DOM elements corresponding to a superhero data row in the table.

3.2 Live Programming

Live programming is implemented by continually running the generalization algorithm on the DOM element under the user's cursor, reverting if the user hovers away and committing when the user clicks. The generated row and column selectors 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.

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:

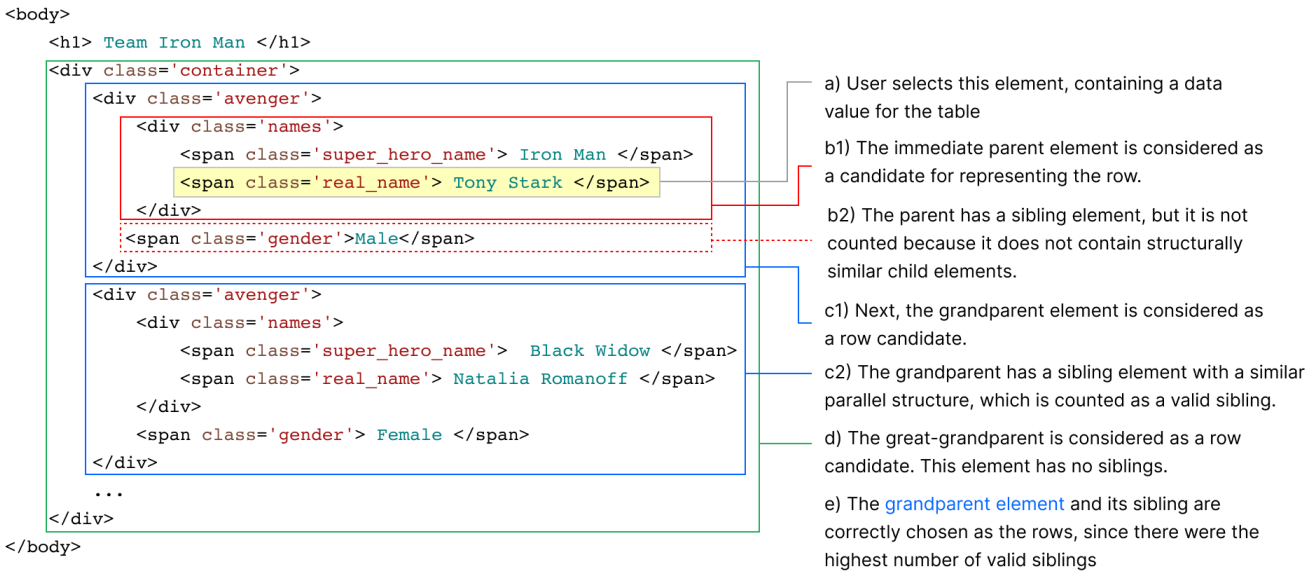

Figure 2: Our system applies a heuristic to identify DOM elements that correspond to rows in the data table.

changing one column's selector does not affect another column's selector. This is not the case for system's in which the output of demonstrations are dependent on each other. 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

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. Take the following sample HTML layout showing two tables of superheros (Team Iron Man and Team Captain America):

```
<body>
  <h1> Team Iron Man </h1>
  <div class='container'>
      <div class='avenger'>
          <span class='super_hero_name'> Iron Man </span>
      </div>
      <div class='avenger'>
          <span class='super_hero_name'>  Black Widow </span>
      </div>
      ...
  </div>
  <h1> Team Captain America </h1>
  <div>
      <div class='avenger'>
          <span class='super_hero_name'> Captain America </
      </div>
      <div class='avenger'>
          <span class='super_hero_name'>  Scarlet Witch </s
      </div>
      ...
  </div>
</body>
```

Without the constraint that row elements have to be direct siblings, the row generalization algorithm could determine the row selector to be *.avenger* because it matches the largest number of parallel structures (has the largest $n_{siblings}$). While this may be the correct result for the task of extraction, it is not for the task of customization. The selector matches all rows across the two tables so the sorting and filtering customizations could place rows in the incorrect table and thereby distort what the tables represent. Because of this, our system currently does not support such DOM structures but we plan to explore the possibility of extracting multiple tables from a website and joining them.

Another structure that our system cannot currently generalize over is one in which column elements are not contained within a row element that is a direct sibling of all the other row elements. Take the following sample HTML layout showing a table of superheros:

```
<body>
  <h1 class='super_hero_name'> Iron Man </h1>
  <span class='real_name'> Real Name: Tony Stark </span>
  <h1 class='super_hero_name'> Captain America </h1>
```

```
  <span class='real_name'> Real Name: Steven Rogers </span>
  <h1 class='super_hero_name'> Black Widow </h1>
  <span class='real_name'> Real Name: Natalia Romanoff </span>
  ...
</body>
```

This page contains one table of data in which rows are made up of an H1 tag (super hero name) and a SPAN tag (real name). For the task of extraction, the DOM elements can be scraped using the shown classes to output a table in which the values of the H1 tags form the first column and the values of the SPAN tag form the second column. For the task of customization, our system would need to know what the row boundaries were in order for customizations such as sorting and filtering to work as expected. In Wildcard, this can be done with a hand-coded adapter by creating artificial row boundaries. In the above DOM structure, an artificial row boundary could be created by representing a row as an H1 tag and the SPAN tag that immediately follows it. These artificial rows would form a single unit which would not distort the website after a sorting or filtering customization. We plan to explore how such artificial row boundaries might be created via demonstration in order to support these structures (which are not uncommon, and can be seen, for example, in HackerNews).

4 DESIGN PRINCIPLES

Below, we discuss the three design principles underlying our work and how they relate to the broader field 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, making it a much less unified environment.

This type of divide between tasks appears in other domains. In data science, workflows revolve between cleaning and using data but this often happens in different environments. The creators of Wrex [5], a 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 wrote their scraping code in order to edit and re-run it. Sometimes, the divide is more subtle. Vegemite [11], a system for end-user programming of mashups, had a unified environment for scraping and augmenting data but separate interfaces for the scraping and augmentation tasks. During a user study, participants reported that "it was confusing to use one technique to create the initial table, and another technique to add information to a new column."

In this work, we have combined scraping and customization into a unified activity within a single environment. 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, 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 augment 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 directly relates to the idea of "in-place toolchains" [1] being an integral part of end-user programming systems.

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 [4], a web scraping program created by demonstration can only be edited through a high-level, block-based programming language called Helena [2] which does not support providing new demonstrations to extend or repair the scraping code. In Vegemite [11], a web automation program created through demonstration can only be edited by editing the text-based representation of the program.

In Wildcard, if a website's hand-coded adapter ceases to function because the website changes, the table may lose its data which means that an end-user's customizations may also cease to function. Furthermore, end-users are powerless to 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 [13] 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 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 some end-user web scraping systems like Rousillon [4], users only get *full* feedback about the program's execution (generalization 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 employs live programming techniques to eliminate this edit-compile-debug cycle by running the generalization algorithm and generating 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. It also populates the table with the scraped data based on the latest demonstration. The highlighting and table population serve to give users a view of how their demonstration has been generalized and what data will be available in the table once scraped.

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 [14], a framework that provides user interface support for programming-by-demonstration systems like FlashExtract [9], and relates to the idea that an important quality of end-user programming is "interaction with a living system" [1].

Unlike text-based commands which are only valid when 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 (incomplete command verus complete command). This allows us to take a small step further by executing 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 by a number of tools.

FlashProg [14] is a framework that provides user interface support for FlashExtract [9], a framework for data scraping by examples. FlashProg's interface provides immediate visual feedback about the generalization and scrapes the matched values in 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, it does not offer a unified environment within a browser for scraping and customizing websites.

Roussillon [4] is a tool that enables end-users to scrape distributed, hierarchical web data. Its interface does not provide *full* live feedback about its generalizations or the values to be scrapped until all the demonstrations have been provided and the generated program has been run. If run on a website it has encountered before, Roussillon 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, Roussillon presents the web scraping code generated by demonstration as an editable, high-level, block-based language called Helena [2]. While Helena can be used to perform more complex

editing tasks like adding control flow, it 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.

Vegemite [11] 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, 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 [10] 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 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" [11] techniques. However, this editing does not extend to the web scraping interface used for demonstrations.

Sifter [7] is a tool that augments well structured websites with advanced sorting and filtering functionality. Like Wildcard, it uses web scraping to extract data from websites in order to enable customizations (sorting and filtering). It performs the web scraping automatically using a variety of heuristics and solicits guidance from the user if this fails. While automatic web scraping seems desirable, it is unclear how useful it is if the goal is customization. Given a row with ten scrapable values, and therefore ten columns, would a user prefer to simply demonstrate the value for the single column they are interested in or un-demonstrate the nine values they are not interested in? This is a question we can only answer after performing a user study.

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 approaches, web scraping in our current implementation is limited to what can be demonstrated. This is problematic if users want to scrape the URL associated with a link element, which is not visible, or only scrape a substring of a value. To solve this, we plan to harness Wildcard's formula language. End-users will be able to use formulas targeted at web scraping to access DOM element properties and attributes. For example, a formula like `=GetAttribute(link_column, 'href')` could be used to scrape URLs of link elements. End-users will also be able to use formulas targeted at data processing. For example, a formula like `=GetSubstring(name_column, 1, 2)` could be used to scrape substrings of text values. Such web scraping and data processing formulas will give end-users some of the power available to programmers that write web scraping code in Javascript which supports a wide variety of DOM access and processing ability.

To assess our design principles, we plan to carry out a broader evaluation of our system through a user study. Furthermore, we plan to incorporate the program viewer and disambiguation features of FlashProg [14] to give users more insight and control into the generalization process.

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

REFERENCES

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

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

[3] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. [n.d.]. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology - UIST '05* (Seattle, WA, USA, 2005). ACM Press, 163. https://doi.org/10.1145/1095034.1095062

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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