Extending a Reactive Expression Language with Data Update Actions for End-User Application Authoring

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

ABSTRACT

Mavo is a small extension to the HTML language that empowers non-programmers to create simple web applications. Authors can mark up any normal HTML document with attributes that specify data elements that Mavo makes editable and persists. But while applications authored with Mavo allow users to edit individual data items, they do not offer any *programmatic data actions* that can act in customizable ways on large collections of data simultaneously or that modify data according to a computation. We explore an extension to the Mavo language that enables non-programmers to author these richer data update actions. We show that it lets authors create a more powerful set of applications than they could previously, while adding little additional complexity to the authoring process. Through user evaluations, we assess how closely our data update syntax matches how novice authors would instinctively express such actions, and how well they are able to use the syntax we provided.

ACM Classification Keywords

H.5.4. Information Interfaces and Presentation (e.g. HCI): Hypertext/Hypermedia- User Issues.

Author Keywords

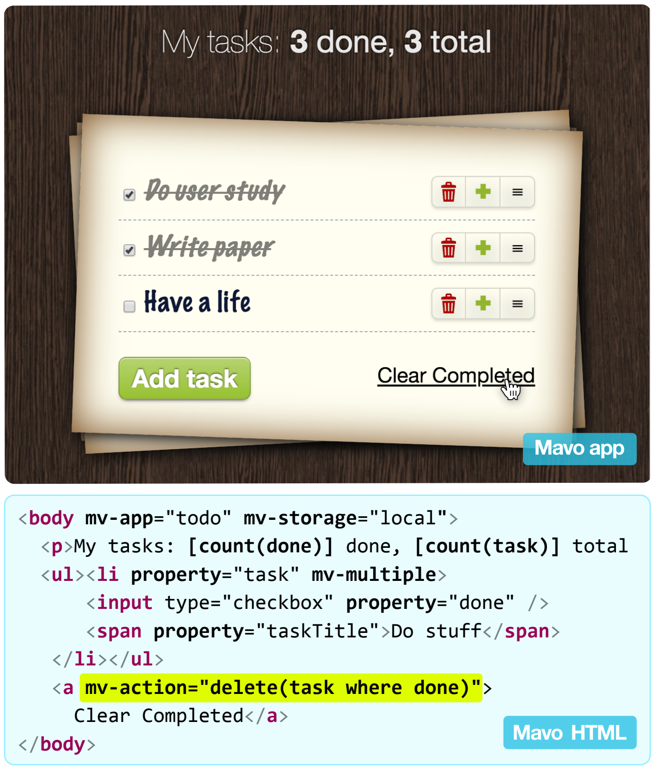
Web design; End-user programming; Information architecture; Semantic publishing; Dynamic Media; Web; Query languages; Data updates; Reactive Programming;.

INTRODUCTION

Many systems and languages exist for assisting novice programmers to manage information, and/or create CRUD applications for this purpose. They range from the well known commercial spreadsheet systems to more complex application builders [8, 5] or simplified declarative languages [15, 2]. These usually generate an editing interface for elementary data manipulations (editing a data unit, inserting items, deleting items) and a mechanism for lightweight reactive data computation. As an example, in the case of spreadsheets the editing

CHI'16, May 07–12, 2016, San Jose, CA, USA

ISBN 123-4567-24-567/08/06...$15.00
DOI: http://dx.doi.org/10.475/123_4



```
<body mv-app="todo" mv-storage="local">
  <p>My tasks: [count(done)] done, [count(task)] total
  <ul><li property="task" mv-multiple>
      <input type="checkbox" property="done" />
      <span property="taskTitle">Do stuff</span>
    </li></ul>
    <a mv-action="delete(task where done)">
      Clear Completed</a>
</body>
```

Figure 1: A fully-functional To-Do app made with Mavo, with a data update action for deleting completed items.

interface is the grid itself, and the reactive computation is the spreadsheet formula.

These edits are typically restricted to direct editing of specific data items by the end user. Affordances may also be provided for aggregating certain kinds of commonly needed mass modifications. A few examples of these would be selecting multiple items for deletion or move, adding multiple rows or columns, or the spreadsheet fill handle. However, the set of potential data mutations is infinite, and it is not practical to predefine controls for every possible case. For more complex automation of data edits, users are typically directed to scripting or SQL queries. Learning a scripting language is generally as hard as learning a programming language, and even if some could argue that SQL queries are easy for a single table, they quickly become complicated when nested schemas are involved, which are represented by multiple tables and foreign keys [6].

Mavo [15] is an HTML language extension for defining CRUD Web applications. In contrast to most systems, instead of asking authors to define a data model and then define a UI over it, Mavo asks authors first to make a static "mockup" of their user interface in HTML, then annotate the HTML with attributes that indicate which elements are actually data that the user should be able to manipulate. Based on this markup, Mavo generates an editing interface for editing each data item manually. Mavo stores its data locally or on a cloud service, and implements a reactive expression language called MavoScript for lightweight computation. Previous work [15] provided evidence that Mavo empowered users with no programming experience to author fully functional CRUD applications. But that implementation of Mavo offered only direct editing of individual data items, while many applications call for richer, programmatic modification of large collections of data simultaneously.

The need for programmatic data updates had become apparent from the early days of Mavo: while a simple To-Do list could be easily implemented in Mavo with a few lines of HTML, clearing completed items was not possible. Mavo did offer controls for deleting individual items, but no way to specify such actions that programmatically delete certain items.

In this work, we extended Mavo with a new HTML attribute, `mv-action`, for specifying programmatic data updates. Its value describes the data mutation as an expression, and it is placed on the element that will trigger the action by clicking. The expression leverages Mavo's existing expression syntax as much as possible, but adds functions that modify the data and are only available within this attribute. We also implemented a few more functions and operators to make it possible to filter and define data in Mavo expressions, to increase the expressiveness of such data mutation expressions.

Our hypotheses are that (a) novice web authors can easily learn to specify programmatic data mutations using this syntax, and (b) that the set of primitives we have chosen is expressive enough to meanigfully broaden the class of data management applications that novices can create without undue complexity. To examine the first hypothesis, we conducted a user study with 20 novice web developers writing expressions for a variety of data mutations, both using their own imagined syntax, as well as ours. We found that the majority of users were easily able to learn and apply these data mutation expressions with 90% of our participants getting two thirds of the questions right on first try, with no iteration. We also found that in many cases, our syntax was identical or very close to their ideal syntax. To examine the second hypothesis, we list a number of case studies of common interactions and how they would be expressed with our data update syntax.

RELATED WORK

Many platforms and systems have been developed over the past few decades to help web authors build web applications. Some of these tools were targeted for web developers with limited programming and database knowledge [4, 16], allowing end users to make programmatic changes to the data using SQL queries. Others were developed for novice web designers who are interested in rapid development of web applications [1, 12, 7], with spreadsheet as a back-end, but with limited or no mechanism to make data updates programmatically.

TODO: Cite Mavo paper for related work on end-user programming systems. Maybe cite elicitation studies?

MAVO DATA UPDATE LANGUAGE

We now describe Mavo's data update language.

The mv-action HTML attribute

Data updates are specified via an `mv-action` HTML attribute, which can be placed on any element. Its value is an expression that describes the action that will be performed. By default, triggering the action is performed by clicking (or focusing and pressing spacebar, for keyboard accessibility), as it appears to be the most common way to invoke an action on most types of GUIs.

Data mutation functions

We extended Mavo's expression language with four new functions that modify the data: `set()`, `add()`, `delete()`, `move()` with the following signatures:

- `set(reference, value)`
- `delete(ref1, [, ref2 [, ref3, ...])`
- `add(collection [, data] [, position])`
- `move(from, to)`

The first three are analogous to the SQL primitives UPDATE, INSERT, DELETE, whereas the latter is a composite mutation (move is essentially delete and then add). These functions are **only** available in expressions specified with the `mv-action` attribute. Regular Mavo expressions remain side-effect free.

We kept these functions minimal, to delegate selection and filtering logic to Mavo expressions. This maximizes the amount of computation specified in a reactive fashion, which we believe is easier for novices to work with (as in spreadsheets).

One thing that sets Mavo's expression language apart is its scoping reference mechanism. Every property can be referenced from anywhere and the value depends on the location of the expression relative to the property in the data tree. Referencing a multi-valued property on or inside a collection item resolves to its local value on that item, whereas referencing that property name outside the collection resolves to *all* values. All operators can be used with both single values and lists of values and are executed element-wise, much like array formulas in spreadsheets. This makes specifying many common expressions concise, often only needing a single property reference, a feature that extends to our mutations as well.

Defining literal data in expressions

To support expressions describing literal complex structures, such as adding objects to a collection, we defined a new `:` (colon) operator, and two new functions: `group()` for objects and `list()` for arrays. By combining these with MavoScript's existing literals, any JSON structure can be specified.

The `:` (colon) operator is used for defining key:value pairs, which the `group()` function can then combine in a single object, akin to the data that Mavo groups produce. The `list()` function produces arrays, like those that Mavo list-valued properties return.

For example, below you can find a JSON object and the corresponding representation in Mavo using the new syntax:

```
{
  "name": "Lea",
  "age": "32",
  "hobby": ["Coding", "Design", "Cooking"]
}
```

```
group(
  name: Lea,
  age: 32,
  hobby: list(Coding, Design, Cooking)
)
```

Other additions

With data mutations, there may sometimes be a need to perform more than one update as part of the same action. Figure 4 demonstrates an example. To facilitate this, we extended MavoScript to support **multiple function calls in the same expression**. These can be separated by commas, semicolons, whitespace, or even nothing at all.

To enable filtering of list-valued properties, we implemented a `where` operator. In imperative languages, such filtering is not as essential, because programmers are expected to collect the target data themselves (e.g. by looping) and assign them to variables that they would then operate on. However, in declarative data update languages (like ours or SQL), filtering is more important, because data is operated on all at once. Since the operator has relatively low precedence, it is possible to use multiple predicates without any parentheses, e.g. `woman where age > 20 and name = 'Lea'`.

EXAMPLE USE CASES

A first part of our argument in this paper is that Mavo's data actions have sufficient power to (with ease) specify a broad range of programmatic data manipulations in applications. To contribute to that argument, we outline a few use cases of common interactions below. None of these use cases can be implemented with the original Mavo alone. In each case, we show the (tiny amount of) source code needed to provide the functionality. Most of this code consists of HTML and original Mavo syntax; we highlight the (even tinier amount of) code leveraging the data action syntax. We first demonstrate how a few lines of Mavo can be used to define a number of frequently used general-purpose UI widgets (that are currently implemented using substantial amounts of Javascript), then present a few specific applications such as the ubiquitous shopping cart.

Generalizing existing Mavo data updates

All Mavo data update controls (except drag and drop, due to the current lack of support for dragging as an update trigger) can be expressed with as data actions very concisely, which opens up many possibilities for UI customization. The following examples assume the updates are modifying a collection with `property="item"`.

Action	Data update Expression
Add new item button *(outside collection)*	`add(item)`
Add new item after current	`add(item)`
Duplicate current item	`add(item, item)`
Delete current item	`delete(item)`
Move up	`move(item, index - 1)`
Move down	`move(item, index + 1)`

Common Interactions and Widgets

One natural class of use cases for data actions is in creating rich new UI widgets. These widgets generally present underlying data from the traditional model in some novel fashion. But the widgets also tend to come with their own internal *view model* describing the state of their presentation. Data actions can be used to control the state of the view model, and thus to manage the widget's data presentation.

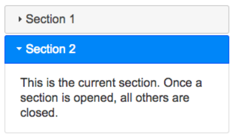
Spinner control

While spinners (a widget that can increment or decrement a number) can now be natively created in HTML5 (`<input type="number">`), this can still be useful if a customized presentation is desired.

```
<input  property="number" />
<button  mv-action="set(number, number+1)" >&#9650;</button>
<button  mv-action="set(number, number-1)" >&#9660;</button>
```

Accordion / Tabs

An accordion or tab set permits a user to choose which of several distinct sections of content should be shown while the others are hidden.



This is the current section. Once a section is opened, all others are closed.

The markup below is for an accordion widget, but the same logic (with different HTML elements and styling) can be used to implement a tabbed view.

```
<details property="prop" mv-multiple open="[open]"
     mv-action="set(open.$all, false) set(open, true)" ">
  <summary property="title"></summary>
  <meta property="open" />
  <!-- content -->
</details>
```

Pagination

The following markup outlines how a working page selection widget could be specified. It uses 10 items per page, but that could also be a dynamic property. It uses the Mavo `mv-value` attribute to generate a dynamic collection of page numbers, then `mv-action` to make them clickable. Then, an expression on each item controls whether it is part of the HTML or not based on the current page.

```
<meta property="curPage" content="1">
<div property="item" mv-multiple
     mv-if="curPage * 10 < index < (curPage + 1) * 10">
  <!-- item properties -->
</div>
<a property="page" mv-multiple
   mv-value="1 .. count(item) / 10"
   mv-action="set(curPage, page)" >1</a>
```

Slideshow / Carousel

A slideshow is essentially pagination with one item per page, frequently used for photos or other large objects.

```
<meta property="current" content="0">
<div property="slide" mv-multiple hidden="[current != index]">
  <!-- content -->
  <button mv-action="set(current, next.index or 0)" ">
  Next</button>
  <button mv-action="set(current, previous.index)" ">
  Previous</button>
</div>
```

Select All

It is common to offer an affordance to simultaneously check or uncheck all items in a list.

```
<button mv-action="set(selected, true)" >Select All</button>
<button mv-action="set(selected, false)" >Unselect All</button>
<div property="item" mv-multiple>
  <input type="checkbox" property="selected" />
  <!-- other content -->
</div>
```

Sorting table by header

Given a data table, it is common to offer the ability to sort by a particular column by clicking on the heading of that column. Mavo provides an `mv-sort` attribute whose value is an expression to be used to sort items in a collection. A data action can then be used to set the value of the sorting expression.

```
<meta property="sortBy" content="">
<table>
  <tr>
    <th mv-action="set(sortBy, 'name')" >Name</th>
    <th mv-action="set(sortBy, 'age')" >Age</th>
  </tr>
  <tr property="person" mv-multiple mv-sort="[sortBy]">
    <td property="name"></td>
    <td property="age"></td>
  </tr>
</table>
```

An alternative way to implement this would be to use the `sort()` function (provided by the same plugin) and overwrite the collection with a sorted version of itself each time the header is clicked:

```
<th mv-action="set(person, sort(person, 'name')" >Name</th>
<th mv-action="set(person, sort(person, 'age')" >Age</th>
```

Adding events to a map

This example positions each collection item over a map by using Mavo's special `$mouse` variables to store the position of the mouse at the time of clicking. Similar logic can be used for adding events to a calendar view.

```
<img src="world-map.svg" mv-action="add(event, hoverPos)" />
<meta property="hoverPos" content="group(
  lat: $mouse.y - 90, lon: $mouse.x - 90)">
<div property="event" mv-multiple
     style="top: [lat]px; left:[lon]px">
  <meta property="lat" />
  <meta property="lon" />
  <!-- other properties -->
</div>
```

Heterogeneous collections

Mavo provides a built in UI for adding items to a *homogeneous* collection, where each item has the same schema so the added item can look like all the others. But this approach breaks down if more than one type of item can be added to the collection. With data actions, heterogeneous collections can be created by having a hidden property for the type of item and separate Add buttons for each type. As an example, we present below the markup for a blog with two types of blog posts: text and picture.

```
<article property="post" mv-multiple>
  <meta property="type">
  <h2   property="title"></h2>
  <img  property="image" mv-if="type = 'image'">
  <div  property="text"  mv-if="type = 'text'">
</article>
<button mv-action="add(post, type: 'image')" ">
  New image post</button>
<button mv-action="add(post, type: 'text')" ">
  New textual post</button>
```

Specific applications

We now turn to specific applications, for which data actions are necessary and sufficient.

E-shop: Add to cart button

This is a specific instance of copying an item from one collection to another, which Mavo offers natively. But nobody wants to force users to do this by dragging and dropping the item. Instead, a dedicated button can do the job.

```
<div property="product" mv-multiple>
  <!-- name, image etc properties -->
  <button mv-action="add(cart, product)" >Add to cart
  </button>
</div>
<div property="cart" mv-multiple>
  <!-- subset of product properties -->
</div>
```

Invoice Manager: Two ways to copy customer details

Another commonly needed functionality is copying data from one item to another, for example copying shipping address to billing address, or this example of copying invoice details.

```
<div property="invoice" mv-multiple>
  Copy customer details from invoice
  #<input property="copy" />
  <button
   mv-action="set(customer, $all.customer where id=copy)" >
  Go</button>

  <button mv-action="add(invoice, customer: customer)" >
    New invoice for this customer</button>
  <!-- invoice properties -->
</div>
```

EVALUATION

Our goal was to design a data update language that would feel natural to novice programmers. We took a two-pronged approach to this challenge. First, we attempted to elicit from users a syntax that they would find natural. Second, we assessed whether a syntax that *we* developed was natural for them to use. We designed our user study with four sections as follows.

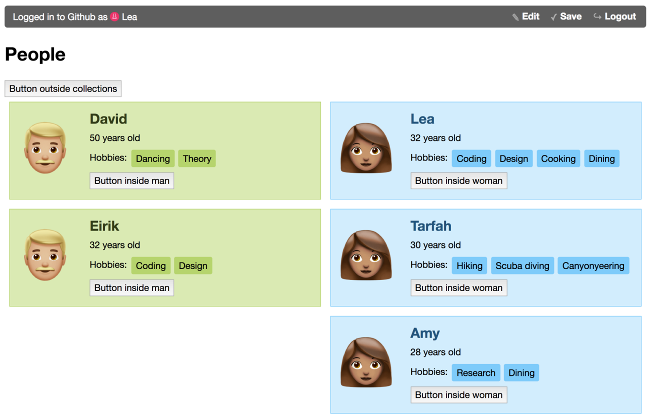


Figure 2: The people application, used in our study for a variety of tasks

	HTML	CSS	JavaScript	JSON
Not at all	0	0	6	4
Beginner	1	1	5	3
Intermediate	5	3	9	8
Advanced	12	11	0	5
Expert	2	5	0	

Table 1: User study participants' familiarity with web development languages.

Preparation

We recruited 20 participants (mean age 36.2, σ =9.25; 40% male, 60% female) by publishing a call to participation on social media and local web design meetup groups. Our participants' skill levels in HTML and CSS ranged from beginner to expert, but all described themselves as intermediate or below in JavaScript. When they were asked about programming languages in general, 11/20 described themselves as beginners or worse in *any* programming language, while 9/20 considered themselves intermediate. In addition, when we asked participants about their experience with various data concepts, only 5/20 stated they could write JSON, 4/20 could write SQL, and none could write HTML metadata (RDFa, Microdata, Microformats). We asked our participants to read through the Mavo Primer[1] and optionally to create a shopping list application (any kind they liked) with Mavo before coming in for the study.

Study Design

User studies were conducted one-on-one, in person. Participants were asked to perform several tasks, which were divided into 4 sections. Every session was limited to an hour and a half; 15 minutes for two sections (the first and the last one) and 30 minutes for the other two. At the beginning of each session, participants were shown a Mavo CRUD application with two collections (men and women) each containing a name, an age and a collection of hobbies (Figure 2). We decided on this schema because it is nested, and the properties have an obvious natural meaning.

[1]mavo.io/docs/primer

First, participants were asked to write expressions that compute counts for five questions of increasing difficulty, starting from the simplest ("Count all men") down to filtered counts (e.g. "count women older than 30", "count women who have 'Coding' as a hobby", etc), which the first Mavo study [15] showed that participants found problematic. Participants were discouraged from iterating on their expressions, and were told we wanted to capture their initial thinking.

The purpose of this part of the study was tri-fold: (a) to assess their understanding of current Mavo capabilities, (b) to verify whether filtered counts were indeed harder, as found in Mavo's user study [15], and (c) to prime them into thinking in terms of declarative functional expressions for the study that was yet to follow. Participants were discouraged from trying their expressions out, and it was communicated to them that the objective of this part was to see their initial thinking, with no iteration.

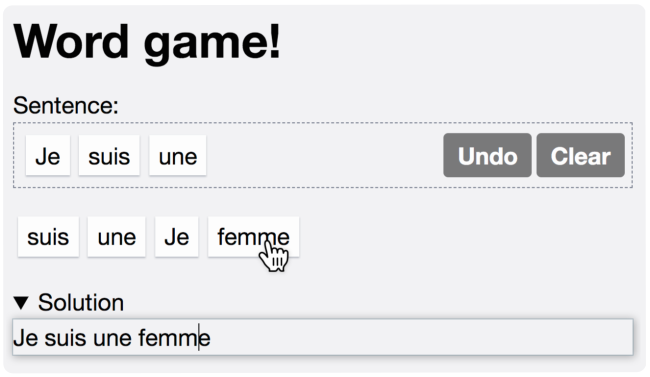
Second, we briefly explained the problem that Mavo data updates are solving, as well as our idea for addressing it on a high level. More specifically, we mentioned the `mv-action` attribute, as well as the `set()`, `delete()`, `add()`, and `move()` functions, but presented this as ideas whose syntax we are not sure about and had not developed yet. We then asked participants to answer 17 data update questions of increasing complexity (Table 2) by writing the syntax that felt more natural to them. They were also encouraged to even use different function names, if they felt more natural to them.

After this stage, we revealed that we had implemented a prototype of this functionality with our best guess at an intuitive syntax, so that they could experiment with it during the study. We then went through a brief tutorial of our syntax prototype (5-10 minutes), after which participants had to answer the same questions, in the same order, but this time using our syntax. After this section, participants were asked to choose 4 questions, one from each action type (set, add, move, delete) and try them out. At this point, researchers would alert them to any mistakes and help correct them, as these were training tasks for the next section of the study.

The final part of the study consisted of two sets of hands-on tasks where participants would try authoring data updates to complete the functionality of two different applications.

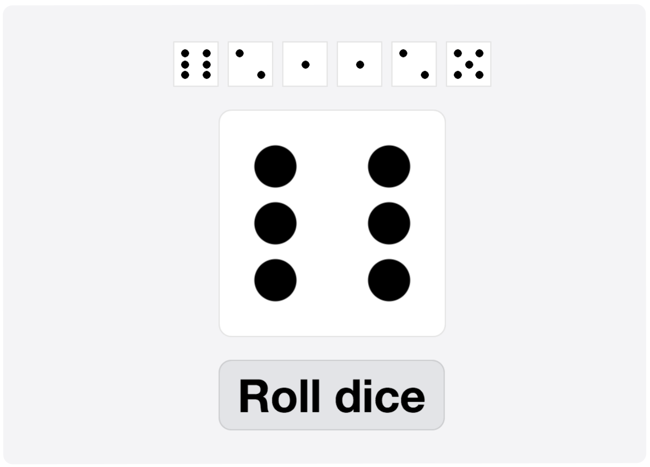
For the first set of hands-on tasks, we created two applications: a DICE ROLLER application (Figure 4) with a history of past dice rolls, and a language learning WORD GAME where users have to click on words in the right order to match a hidden sentence (Figure 3). Participants were randomly assigned to one of the two. The WORD GAME consisted of three tasks for its three data update calls (clicking on words, Clear, Undo). The dice roller also consisted of three tasks: First to roll a random dice, then to add it to the history, and lastly to prevent the current dice roll from showing up in the history.

The next set of tasks was the same for all users and centered around a shopping list application, either the one they made,



```
<div class="[if(join(orderedWord, ' ') = solution, correct)]">
      <span property="orderedWord" mv-multiple></span>
      <button mv-action="delete(orderedWord)">Clear</button>
      <button mv-action="delete(last(orderedWord))">Undo</button>
</div>
<span property="word" mv-multiple
       mv-value="shuffle(split(solution))"
      mv-action="add(orderedWord, word)"></span>
```

Figure 3: The Word Game application with its solution



```
<div property="diceHistory" mv-multiple
      class="dice-[diceHistory]"></div>
<div property="dice" class="dice-[dice]">6</div>
<button mv-action="add(diceHistory, dice),set(dice, random(1,6))">
Roll dice</button>
```

Figure 4: The dice rolling application with its solution

or our template. The two actions required to complete it were adding a button to copy the current item to the list of common items, and to be able to click on common items to copy them to the shopping list.

For all hands-on tasks, participants were given the HTML, CSS and (original, non-data-action) Mavo markup, and only had to add the `mv-action` attributes to complete their functionality.

After finishing all tasks, participants were asked a few questions about their experience in the form of a brief semi-structured interview, completed a SUS [3] questionnaire, and a few demographics and background questions.

RESULTS & DISCUSSION

Counting questions

All participants' answers to the simple counting questions were correct, even when they had to count a scoped property (counting all hobbies from outside both collections of men and women). Also, they seemed to have no trouble with filtered aggregates like `count(age > 3)` with 17/20 people getting them right, and the remaining three making only minor syntax errors.

Instead, what participants found hard was disambiguating between scoped properties with the same name across two collections (i.e. getting only women's ages or men's hobbies). Mavo (like SQL) uses dot notation for this (i.e. `woman.age` only returns women's ages), which only 8/20 participants used. However, this is likely related to there being no example of this in the Mavo Primer, so we did not consider failure on this topic to be a sign of poor understanding of the original Mavo functionality.

Free-form Syntax

In this part of the study, we wanted to explore what syntax participants found natural, with the only constraint being that they had to use the four functions (set, add, move, delete). This constraint was introduced to put participants in the mindset of writing expressions instead of natural language. They were even encouraged to use different function names if they wished to, and 6 did so at least once (half of which inconsistently).

Despite emphasizing that constraint, 6/20 participants did not use any functions for answering at least one of the 17 questions but wrote statements instead (such as `age = age + 1`) and 4 more used a hybrid approach, with some of the parameters being outside the function call, such as `add(woman) name=Mary age=30`.

The median time each participant spent answering each question was 28.5 seconds.

Scope

We mentioned in the Introduction that one feature of Mavo expressions is the ability to use any property name anywhere and either get the local value or all values depending on the placement of the expression, enabling very concise references for the common case. Thus, for example, delete(man) used "inside" a particular man in a collection of man objects would delete only that man, while delete(man) used "outside" the collection would delete *all* those man objects.

However, in their own syntax, many subjects wanted to make the distinction between current item and all items explicit. 8/20 used an explicit keyword or function for all items (e.g. `man.all`) at least once, and 8/20 used an explicit keyword or function for the current item, such as `woman.current` or `this`. Only 3 participants did both. It is interesting to note

#	Question	Type	▼
1	Delete all men	delete	
2	Add new man (with no info filled in)	add	
3	Delete all people	delete	
4	Add a new man and a new woman	add	
5	Delete current man	delete	
6	Make current man 1 year older	set	
7	Make everyone 1 year older	set	
8	Set everyone's name to their age	set	
9	Delete women older than 30 years old	delete	✓
10	Move the current woman to the collection of men	move	
11	Add a woman with the name "Mary" and age of 30	add	
12	Add a woman with the name "Mary" and age of 30 to the beginning of the women collection	add	
13	Delete "Dining" as a hobby from everyone	delete	✓
14	Rename every man with age > 40 to "Chris"	set	✓
15	Move the current woman to the beginning	move	
16	Change the age of the woman named "Mary" to 50	set	✓
17	Move all men to the collection of women	move	

Table 2: All 17 data manipulation questions. The third column indicates whether filtering was needed to answer the question.

that **none of them followed their own referencing schemes consistently**, using these explicit references only in some of the questions or some of the arguments, while using plain property names elsewhere. This may indicate one reason why this referencing scheme is useful: it eliminates error conditions.

More work is needed to understand exactly why our subjects attempted this more verbose language when the more concise one would work. Based on participant answers to probing questions about this, it seemed that the survey format may have played a role: they were writing their answers in text fields that were separate from the HTML, so it was hard to imagine the context of their expression. In that setting, it may have been jarring to write the same expression as an answer to completely different questions (e.g. "Delete all men" and "Delete current man" are both `delete(man)` with our syntax). Perhaps if they'd been writing Mavo expressions inside actual HTML, the disambiguation through context would have eliminated the desire to disambiguate through syntax.

Another possibility is that novice programmers *prefer* verbosity. Pane et al. [11] showed that 32% of non-programmers constructed collections by using the keywords every or all (which is similar to the percentage we observed). The use of such verbose syntax could be seen as a form of commenting, adding clarity over more concise code. It would be easy to provide syntactic sugar to allow these more verbose, explicit references to all items versus the current item in the future. In fact, Mavo already defines a special `$all` variable that works in a similar way (either by itself, or as a property), although we did not teach them about this.

We also observed the reverse, of users trying to be *more* concise than Mavo. In particular, 7/10 participants indicated that the target of their action is the current item by **omitting** a parameter, such as writing `delete()` for deleting the current item or `move(man)` for moving the current woman to the collection of men. These are potentially dangerous ambiguities; for example, while one could apply consistent syntax to `delete()` of deleting the immediately containing item, an author may actually have intended to delete a higher up ancestor.

Underspecified expressions

While there is no objective right or wrong regarding certain syntax choices such as which punctuation to use, there are other cases where the user's chosen expression must be regarded as incorrect because it does not specify everything necessary for the operation. However, even in those cases, it is hard to be certain that there is a logic error at play. Are the missing parameters actually missing, or did the participant have a clever heuristic in mind for inferring them? And if not, is it a logic error, or merely a slip? In this section, we describe some of the most common patterns of (ostensibly) underspecified expressions that we observed.

By far the most common such pattern was using `delete(<PREDICATE>)` with no mention of which item(s) should be deleted. For example, `delete(woman.age > 30)` for deleting women older than 30, or `delete(hobby='Dining')` to remove the hobby "Dining" from all people. 18/20 participants did this at least once, and 10 did so in both of the conditional delete questions in Table 2 (Q9, Q13). One possible interpretation would be that the set targeted for removal is specified by the first token in the expression—e.g. "woman" and "hobby" in the above examples. But this seems likely to be fragile in general use—for example, does the second expression above try to delete all "Dining" hobbies, or does it try to delete all women who have dining as a hobby?.

Another common pattern was using `set(age+1)` to increment all ages with 15/20 participants using a variation of this syntax. Note that this is consistent with the proposed interpretation above, that the update target is the first named token. Interestingly, none underspecified their expression for setting all names to the corresponding ages, where there were two properties at play. Furthermore, none of the participants realized the inconsistency when they answered the latter and did not think to back and change their answer to the former. Inquiring a subset of participants about this after they had entered their answers revealed that some of them thought of `age+1` as an increment operator (like C's `age++`) whereas for others, omitting the property to be modified was a slip.

Both of the aforementioned patterns may indicate a distaste for parameter repetition, which is on par with how natural language works: "parameters" are only explicitly specified when different and are otherwise implied. **TODO: cite?**

5/20 participants wrote their expressions as if the "name" property had a special meaning, i.e. was an implied primary key. For example they would use `Mary` as a way to refer to the person that has name="Mary", without specifying "name" anywhere in their expression. This did not seem to correlate with a lack of (self-reported) programming skill, as only one of them had not been exposed to programming at all. It is unclear whether this has to do with the word “name” itself, or with the fact that names were indeed unique in the data we gave them.

Using objects as numbers, for example using `delete(woman > 30)` or `set(man + 1)` without specifying “age”, was relatively common with many participants attempting it at first, and 4/20 submitting their answers with it. In many cases this turned out to be a slip, but two participants could describe a consistent mental model of how this would work: it would automatically operate on all numeric properties! In Pane et al. [11], 61% of non-programmers modified the object itself instead of its properties, which is even higher than the percentage we observed.

Argument separation

While commas are probably the most widespread mechanism of separating function arguments, they did not appear to be very natural to our subjects. 7/20 did not use any commas at all, but instead separated their arguments by other symbols, or even plain whitespace. 5/20 only used commas only for repetition of the same type of argument (e.g. `delete(man, woman)`). From the remaining 8 subjects, only 2 used commas exclusively to separate arguments, while the rest combined commas with other separators that were more related to the task at hand.

16/20 subjects used “=” to separate arguments at least once, most commonly in `set()`. 9/20 used “to”, primarily in `set()` and `move()`. Other separators (whitespace, colons, and parentheses) were used by 3 people or fewer.

Sequencing function calls

Only 8/20 participants used separate function calls in their expressions (such as `add(man) add(woman)`. The rest instead tried to express the entirety of the action via arguments of one function call, even when this was inconsistent with their later responses.

This may relate to the fact that in spreadsheets, expressions have no side effects and only produce one output, so there is never a need for multiple adjacent function calls. Therefore, using more than one function call may feel foreign and unnatural to these users.

Filtering

Four of our questions required filtering on a collection (cases where the corresponding SQL query would need a WHERE or HAVING condition regardless of the schema used to represent this nested structure). Half of our participants defined a language-level filtering syntax, such as the keywords `if` or `where`, or parentheses (e.g. `woman(age > 30)`). 6/20 expected that the data mutation functions would allow for an extra argument for filtering (akin to our `setif()`/`deleteif()`/`addif()`/`moveif()` functions, which we did not mention to them).

5/20 seemed to expect that predicates would act as a filter of the closest collection item and used them in that fashion consistently, across all filtering questions. For example, they expected that `man.age > 40` would return a list of men whose age was larger than 40, and wrote expressions like `set((man.age > 40).name to "Chris")` for Q16. However, in Mavo currently the inner expression returns a list of booleans corresponding to the comparison for each man.

Relationship to current syntax

Participant free-form syntax would have produced the desired result today with no changes in 4.35 answers on average (σ = 2.16) and in 8.6 answers on average (σ = 2.06) with minor changes (different symbols or removing redundant tokens)

Current Syntax

In this part of the study, we revealed our syntax prototype to participants and asked them to answer the same questions again, but this time using our syntax, to test the learnability and usability of our current design. Participants were not allowed to test their expressions, and were discouraged from iterating because we wanted to capture their initial thinking. Therefore, correct answers in this section are essentially equivalent to participants getting the answer right **on first try** and with no preceding training tasks.

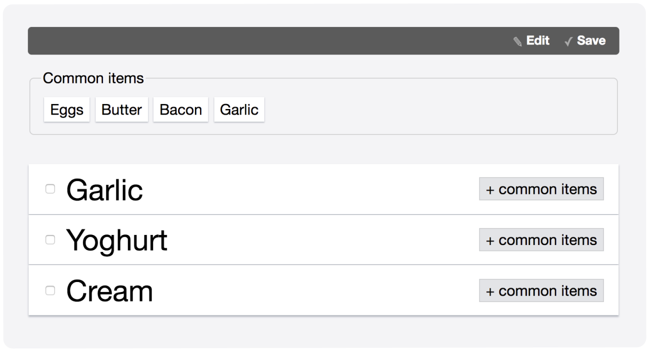
Overall, 11 out of 17 questions had a correctness rate of 75% or above with 8 of them (Q1-3, Q5, Q8, Q9, Q10, Q17) having a correctness rate of 90% or above, i.e almost every participant got them right on first try.

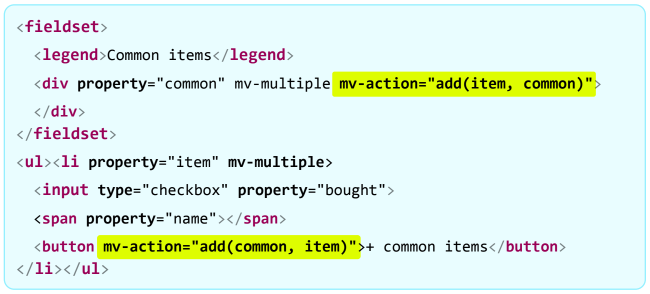
The most prominent patterns from the previous step persisted in this part as well, though to a lesser extent. 7/20 participants remained unable to use a sequence of two separate function calls to answer Q4 (adding both a man and a woman to the collections), and instead wrote `add(man, woman)` or a variation thereof. We did cover this issue in the tutorial prior to the evaluation, but it may not have stuck. Curiously, based on later answers, they all seemed to understand that the second parameter of `add()` was for initial data, but none realized the inconsistency. Similarly, 4/20 participants still used `set(age + 1)` to increment ages, 2/20 used objects as numbers, and 8/20 used `delete(<PREDICATE>)`.

Almost all failures in the *“add with initial data”* type of questions (Q11-12) were related to grouping the key-value pairs or incorrectly using equals signs (`=`) instead of the correct colons (`:`) to separate them.

There were two questions that asked them to delete items with a filter, with vastly different success rates. 18/20 participants (90%) got Q9 correct, while only 9/20 (45%) got Q13 right, despite the superficial similarity of the two questions. Participants were having a very hard time using `hobby` twice in Q13 (The correct answer is `delete(hobby where hobby = 'Dining')` and even those that got it correctly hesitated a lot before writing it.

By *far* the hardest questions were Q14 and Q16, where participants had to filter on one property and set another. Only 7/18 (38.89%) of participants answered these questions correctly. However, every participant knew which function to use, and almost every participant used `where` correctly for filtering, but were then stuck at where to place the property they were setting. As an example, for Q14, they wrote something along the lines of `set(man where age > 40, "Chris")` and were then stuck about where to put `name`. The correct syntax in this case





```
<fieldset>
  <legend>Common items</legend>
  <div property="common" mv-multiple mv-action="add(item, common)">
  </div>
</fieldset>
<ul><li property="item" mv-multiple>
  <input type="checkbox" property="bought">
  <span property="name"></span>
  <button mv-action="add(common, item)">+ common items</button>
</li></ul>
```

Figure 5: The shopping list application with its solution (for participants who did not bring their own)

(if using the `where` operator) would have been `set(man.name where age > 40, 'Chris')`, which is indeed confusing as one would expect the property being set to be grouped with its value, not with the filtering predicate.

Hands-on Tasks

16/20 of our participants completed the hands-on section of the study. Half were randomly assigned to the Dice Roller application (Figure 4), and the rest were assigned to the Words Game application (Figure 3). 13 of those also completed the Shopping List application tasks.

Dice Roller Application

All 8 participants solved the first task and second tasks correctly and were able to display a random dice roll within a median time of **55 seconds** and to display it in the history within a median time of **70 seconds**. 5/8 and 3/8 respectively solved these tasks correctly on first try. Despite eventually getting it right, 5/8 participants hesitated before using multiple function calls in `mv-action`, even if they had answered Q4 with two function calls in the survey.

Even though the second task was carefully worded to avoid implying a particular order of the two operations, all 8 participants put the `add()` after the `set()` they had written in the first task. This resulted in the current die being placed in the history as well as showing in the main display. Doing the opposite order would have rendered the third task redundant, yet nobody realized this. Furthermore, only 1 participant was able to solve the third task and prevent the current dice from appearing in the history. All they had to do for this task is to use `add()` before `set()`, i.e. swap the order of the two functions they had written. This would add the dice to the history *before* they replace its value with a new random value. **None** of the other 7 participants was able to figure out why this was happening, nor how to solve it. Some participants were under the impression that the multiple function calls are somehow executed in parallel, which appears to be a common misconception of novice programmers [10]. This appears to be a general failure of computational thinking, rather than something specific to Mavo.

Words Game

This proved to be a substantially easier task than the dice roller. 6/8 participants were able to write all three expressions, within median times of 220, 43, and 115 seconds respectively.

For the first task, a common mistake (3/8 participants) was to use `move()` instead of `add()` to copy the clicked word into the sentence. Even after realizing their mistake, they were ambivalent about using `add()`.

Shopping List

13 subjects carried out the Shopping List tasks, 6 on their own shopping list application and 7 on ours. Their task was to implement functionality for copying items to and from a "Common Items" collection.

Almost all participants were able to carry out the shopping list tasks, with only 1/13 failing the first task (using our app) and 3/13 failing the second task (2 using our app, 1 using their own). It took slightly longer for participants using their own app to get started on the first task with a median of 133 seconds vs 55 seconds for those using our app. By the second task the difference had been eliminated (median of 55 vs 50 seconds).

Three participants were confused about whether to use `move()` or `add()` to copy the shopping list item to the common items, but after trying, they quickly figured it out.

System Usability Scale (SUS)

At the end of each session, subjects rated their subjective experience on a 7-point SUS scale with 10 alternating positive and negative questions. The answers were then coded on a 5-point scale and the SUS score was calculated according to the algorithm in [3]. We removed one participant who had selected the exact same answer ("Agree") across all 10 questions (positive and negative), which indicates lack of attention to the questions and is a common problem with SUS.

Our raw SUS score was 76.3 ($\sigma_{\bar{x}} = 2.43$), which is higher than 77.5% of all 446 studies detailed by Sauro [14] Our raw Learnability and Usability scores as defined by Lewis and Sauro [9] were 78 and 69.7 respectively.

General Observations

The overall reactions to the data mutation functions ranged from positive to enthusiastic. Several participants remarked on the perceived intuitiveness of the syntax. One participant answered several questions on the survey in one go, without looking at the documentation, then paused and said "it's so intuitive, I don't even need to look at the docs!". Many other participants remarked on expressiveness. "it is very easy to do complex things!", as one of our participants phrased it.

Most participants described our data update actions as easy, even those who made several mistakes. One user said "This is very application-centered, a page that can actually do something!". Another user commented on the data mutation functions saying "I think they are very useful, easy, and approachable", and another user said "it is definitely more accessible than having to program, so that's pretty cool". Another one said "They are easier and quicker to make things without worrying about technicality. It is very easy to use"

As with the first Mavo study, many participants liked being able to add these data update functions to the HTML as opposed to editing in a separate file and/or in a separate language. One user said "Interesting to be able to do these things from the HTML!" and another user said "It is interesting!...being able to do this in HTML, I was able to use it pretty easily, once I knew what functions there were and the syntax it has it was very easy."

They also liked the fact that they can build applications with functionality that would usually be implemented in JavaScript. One user said "This is easier than JavaScript! if I wanted to do something more complicated I would be frustrated to use JavaScript cause I am not good at it, but this is easier". Another user said "It is easier and quicker to make things without worrying about technicality. It is very easy to use".

Several participants commented positively on the `where` operator. "the where syntax is like natural language, I did not expect it to be there and written as if I am saying it".

FUTURE WORK

Usability of Mavo property references

While Mavo's scoping mechanism allows expressing common things concisely, there is *some* evidence from our user study that certain users may prefer a more explicit referencing scheme. However, due to the survey format being a confound, as it removed the HTML context around each expression, we could not study the usability of this reference overloading in this study. More work is needed to determine whether this reference mechanism helps or hinders novices.

Improving the learnability of our syntax

We plan to apply many of the user study findings to improve our syntax and make it more intuitive and readable. Many participants wanted to use a `to` keyword, and we can easily add this and other potentially helpful keyword arguments. Several participants were confused about the `group()` function, what it does, when it is needed, so we will examine whether it is possible to design the language in such a way that `group()` is not required, possibly by using a variable number of arguments in `add()` or by requiring plain parentheses instead of `group()`.

We may decide to special case certain patterns to match user expectations, such as using predicates as the sole argument in `delete()`, rewriting `set(a = b)` as `set(a, b)`, and not requiring repetition in `where` when filtering collections of primitives (e.g. expanding `hobby where 'Dining'` to `hobby where hobby = 'Dining'`).

We need to significantly improve the syntax for questions like Q9 and Q13 (filtering on one property and setting another), since our user study indicated a clear problem with the current syntax. Perhaps exposing the `setif()` etc functions we have implemented would be sufficient or otherwise modifying the syntax of `set()`.

More Triggers

Currently, our data actions are triggered by clicking (or keyboard activation for keyboard users). In the future we are planning to add the ability to specify different triggers, such as double clicking, keyboard shortcuts, dragging, or mousing over the element.

We are considering doing this by adding an `mv-action-trigger` HTML attribute whose value would be an "event selector", as defined by Satyanarayan et al. [13]. Event selectors facilitate composing and sequencing events together, allowing users to specify complex interactions very concisely. Furthermore, since many of our target users are familiar with CSS selectors, they might be able to transfer some of that knowledge.

A different approach would be to also use an function-based syntax as the value of this attribute, and expose event-related information as declarative variables . Mavo already does this to a small degree, by exposing special properties[2], such as `$mouse.x`, `$mouse.y`, `$hash`, `$now` and others, which update automatically, even when no data has changed. We could expand this vocabulary to expose more event information (such as which key is pressed), which would also be useful for expressions in general. This approach would also make it possible to use data changes as triggers. While powerful, this could easily result in cycles, which may confuse novices.

Data updates on computed properties

Mavo supports a concept of "computed properties", i.e. properties whose value is an expression. Currently, applying a data update to a computed property is silently ignored. However, this does not make for a great development experience. Furthermore, there are valid use cases where one may want to temporarily replace or "freeze" the value of an expression. For example, a stopwatch, with a Start/Resume and a Pause button. More work is needed to determine the best way to address these cases. We *could* allow data updates to work with computed properties, and just remove the expression (essentially freezing them in time), but it is unclear how this could be reversed. There could be a special syntax to declare that the value to set is an expression, and not a literal value (essentially getting our HTML authors to declare functions), but since this is a higher level of abstraction, it may prove to be confusing for our target users.

CONCLUSION

This paper presents an extension to Mavo's reactive expression language, for specifying programmatic data updates that are triggered by user interaction. Our user study showed that HTML authors can quickly learn to use this syntax to specify a variety of data mutations very concisely, significantly expanding the set of possible applications they can build, with only a little increase in language complexity.

[2]mavo.io/docs/functions?role=special

REFERENCES

1. Edward Benson and David R Karger. 2014. End-users publishing structured information on the web: an observational study of what, why, and how. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, 1265–1274.

2. Edward Benson, Amy X. Zhang, and David R. Karger. 2014. Spreadsheet Driven Web Applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 97–106. DOI: `http://dx.doi.org/10.1145/2642918.2647387`

3. John Brooke and others. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

4. Stefano Ceri, Piero Fraternali, and Aldo Bongio. 2000. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks* 33, 1-6 (2000), 137–157.

5. Kerry Shih-Ping Chang and Brad A Myers. 2017. Gneiss: spreadsheet programming using structured web service data. *Journal of Visual Languages & Computing* 39 (2017), 41–50.

6. HV Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. 2007. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 13–24.

7. David R Karger, Scott Ostler, and Ryan Lee. 2009. The web page as a WYSIWYG end-user customizable database-backed information management application. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 257–260.

8. Keith Kowalzcykowski, Alin Deutsch, Kian Win Ong, Yannis Papakonstantinou, Kevin Keliang Zhao, and Michalis Petropoulos. 2009. Do-It-Yourself database-driven web applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR'09)*. Citeseer.

9. James R Lewis and Jeff Sauro. 2009. The factor structure of the system usability scale. In *International conference on human centered design*. Springer, 94–103.

10. Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2011. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education* 21, 1 (2011), 57–80.

11. John F Pane, Brad A Myers, and others. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies* 54, 2 (2001), 237–264.

12. Dennis Quan, David Huynh, and David R Karger. 2003. Haystack: A platform for authoring end user semantic web applications. In *The semantic web-ISWC 2003*. Springer, 738–753.

13. Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 659–668.

14. Jeff Sauro. 2011. *A practical guide to the System Usability Scale: Background, benchmarks & best practices*. Measuring Usability LLC.

15. Lea Verou, Amy X Zhang, and David R Karger. 2016. Mavo: creating interactive data-driven web applications by authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 483–496.

16. Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. 2008. Wysiwyg development of data driven web applications. *Proceedings of the VLDB Endowment* 1, 1 (2008), 163–175.