Styling in Espalier: A Spreadsheet Tool for Manipulation of Structured Data

by

Willow Jarvis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

© Massachusetts Institute of Technology 2019. All rights reserved.
Styling in Espalier: A Spreadsheet Tool for Manipulation of Structured Data

by

Willow Jarvis

Submitted to the Department of Electrical Engineering and Computer Science on February 1, 2019, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

The spreadsheet is a commonly-used programming tool, useful for a wide variety of data-driven tasks. They, however, lack the capability of robustly representing and manipulating structured data. Espalier is a new spreadsheet-based programming paradigm which uses a powerful scoped-splits model that more easily handles structured data. This thesis focuses on adding styling capabilities to Espalier in order to allow users to build responsive websites. Overall, the thesis makes two contributions: First, we create a styling menu and implement the set of styling parameters associated with allocations and spans listed in the menu. Second, we style the design view of our sheets using those parameters to highlight the underlying structure. In the following chapters, we will describe our styling implementation, how it is utilized in its design view, and provide two applications to demonstrate its capability, flexibility, and ease of use.

Thesis Supervisor: Daniel J. Jackson
Title: Professor
Acknowledgments

First of all, I would like to thank Professor Jackson for this wonderful opportunity to work on the project and for his kindness, guidance and insights throughout. I would also like to thank Ph.D. student Matt McCutchen, for his advice, patience, and immense knowledge on the structure of Espalier.

I would also like to thank the Gates Millennium Scholarship Program, funded through the Bill & Melinda Gates Foundation, for supporting first-generation students like me to attend the colleges of our dreams.

Additionally, I would like to thank my first undergraduate research supervisor, Mohammad Ghassemi, who guided me into field of computer science.

In the end, I would like to thank my parents, for their unconditional love and support, and especially my mom, for teaching me the importance of an education and for encouraging me to pursue my dreams. Finally, I would also like to thank my friends for the fond memories we have shared and Pranam for his support and encouragement throughout.
Contents

1 Introduction .................................................. 13
   1.1 Motivation Behind Espalier ................................. 13
   1.1.1 Espalier ................................................. 14
   1.2 Related Works ............................................. 14
   1.3 Thesis Overview ......................................... 15

2 Structures and Definitions ................................ 17
   2.1 Custom Layout ............................................ 17
   2.1.1 Split, Scope, and Span ................................. 18
   2.1.2 Allocations ............................................. 21
   2.1.3 Why Scoped Splits ..................................... 22
   2.1.4 Default View, Design View, and Unprivileged View .... 22
   2.2 Espalier as an App Builder ................................. 23

3 Styling Properties ............................................ 25
   3.1 Overview .................................................. 25
   3.2 The Effect of Design View Transformation ................. 25
   3.3 Styling Properties ....................................... 27
   3.4 The Styling Menu ........................................ 29
   3.5 Store Example with Styling Properties .................... 31

4 Design View Styling .......................................... 35
   4.1 Design View Transformation ............................... 35
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Custom Layout</td>
<td>20</td>
</tr>
<tr>
<td>3-1</td>
<td>Allocation Style Properties</td>
<td>26</td>
</tr>
<tr>
<td>3-2</td>
<td>Span Style Properties</td>
<td>27</td>
</tr>
<tr>
<td>3-3</td>
<td>Allocation Part of Styling Menu</td>
<td>29</td>
</tr>
<tr>
<td>3-4</td>
<td>Text Sub-menu</td>
<td>30</td>
</tr>
<tr>
<td>3-5</td>
<td>Border Sub-menu</td>
<td>31</td>
</tr>
<tr>
<td>3-6</td>
<td>Store Application</td>
<td>34</td>
</tr>
<tr>
<td>4-1</td>
<td>Login Page in Design View</td>
<td>38</td>
</tr>
<tr>
<td>4-2</td>
<td>Allocation to Cover Collection Name</td>
<td>39</td>
</tr>
<tr>
<td>5-1</td>
<td>Result of the Old Sizing Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>5-2</td>
<td>Sizing Section of the Styling Menu</td>
<td>45</td>
</tr>
<tr>
<td>5-3</td>
<td>Selection Merges over Multiple Spans on the Column Axis</td>
<td>46</td>
</tr>
<tr>
<td>5-4</td>
<td>A Sizing Example</td>
<td>49</td>
</tr>
<tr>
<td>5-5</td>
<td>The Previous Sizing Example With Annotated Allocations</td>
<td>51</td>
</tr>
<tr>
<td>5-6</td>
<td>Product Page in Default View</td>
<td>63</td>
</tr>
<tr>
<td>6-1</td>
<td>Original Facebook Homepage</td>
<td>66</td>
</tr>
<tr>
<td>6-2</td>
<td>Large Screen</td>
<td>67</td>
</tr>
<tr>
<td>6-3</td>
<td>Medium Screen</td>
<td>68</td>
</tr>
<tr>
<td>6-4</td>
<td>Small Screen</td>
<td>69</td>
</tr>
<tr>
<td>6-5</td>
<td>Comparison of the Menu Bar on a Small Window Width</td>
<td>70</td>
</tr>
<tr>
<td>6-6</td>
<td>Comparison of the Menu Bar on a Further Decreased Window Width</td>
<td>70</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Settings, Defaults, and Constraints ........................................ 44
5.2 Helper Sizes Definition ...................................................... 48
5.3 Inner and Outer Default Sizes Computed .................................. 50
5.4 Helper Sizes After Adjusting for Merged Allocations .................... 53
5.5 Helper Sizes at End of Bottom-up Round .................................. 56
5.6 Helper Sizes After Assigning Sizes after Processing Root Row Span . 57
5.7 Helper Sizes after Split1 is Processed ..................................... 58
5.8 Helper Sizes after Split2 is Processed ..................................... 58
5.9 Helper Sizes after Split3 is Processed ..................................... 59
5.10 Helper Sizes after all Spans are Processed ................................ 60
Chapter 1

Introduction

As programmers, we often think of Python or Java as the most commonly-utilized programming language. However, Microsoft Excel, a spreadsheet application, is surprisingly the more widely-used programming language. In fact, a study in 2005 found that out of about 90 million people who use computers at work in the United States, nearly 50 million utilize spreadsheets and databases, yet only 12 million call themselves programmers [1]. Furthermore, due to the popularity of spreadsheets, developers often utilize them to provide a way for users to download or upload application data, or even to interact with persistent data such as Google Forms [3]. Overall, spreadsheets have evolved into universal tools for structuring data in order to store, display or interact with them. Thus, they have the potential to do even more than what they are currently offering.

1.1 Motivation Behind Espalier

Almost all organizations possess tasks that involve collection and manipulation of structured data subject to certain rules. For example, a school may need to schedule parent-teacher conferences such that every student’s parent may meet one teacher when that teacher is available. These structured data are often relational with some hierarchical structuring [3]. We need a robust method to store and manipulate such relational structured data. Unfortunately, conventional spreadsheets are not built to
fully support such data.

1.1.1 Espalier

Espalier is a new programming paradigm that rethinks the computational model of spreadsheets. By using a spreadsheet model that treats the data in a structured manner so that users can view and update their data subject to the constraints specific to an application, it fully captures and represents structured data. Thus, with Espalier, we hope to provide developers with an application that achieves the optimal balance between its capability and its complexity [2]. This thesis specifically focuses on allowing users to style their spreadsheets directly, thus enabling Espalier to be used as an application builder.

1.2 Related Works

There are existing data-centric applications, such as HyperCard, FileMaker, App2You, QuickBase, and Zoho, that have already been built to handle such structured data. The above existing applications, however, have not gained traction in usage from companies or general users. We suspect that this is because these applications have not provided a natural balance of simplicity and generality to attract stable users. Instead, most either continue to use general-purpose application frameworks such as Rails and Meteor, which are too technically complicated for such data, or utilize a combination of tools such as email, spreadsheets and online forms that, while easy to use, are not complex enough to fully support the demands of such data. Overall, these spreadsheets and online forms have a model that treats the two-dimensional grid as largely unstructured [2]. Furthermore, the existing data-centric applications do not always support stylistic customization. The ones that do, such as QuickBase, for example, mostly have built-in templates that come already stylized for different components and are, therefore, very limited in styling of different sub-components.
1.3 Thesis Overview

The next chapter of this thesis provides an overview of different components in Espalier and any necessary knowledge on the structures of Espalier in order to fully understand scope and depth of the styling involved in this thesis. Chapter 3 characterizes the styling properties we support in the styling menu. Chapter 4 demonstrates how the design view is styled using those properties. Chapter 5 describes how sizing of the spans is specified and computed. Finally, Chapter 6 evaluates the styling by replicating a real-world example: the Facebook homepage.
Chapter 2

Structures and Definitions

Previously in Espalier, we used a fixed layout which used nested tables and supported hierarchical data well. To improve the support relational data as well as reducing unnecessary nesting, we built new spreadsheet UI to provide users with greater control of the layout and appearance of the sheet, which is a major aspect of the appeal of conventional spreadsheets. We call this new layout custom layout. The focus of this thesis is on styling the custom layout by allowing users to specify properties such as colors, fonts, widths and heights, and sizing. We hope to make the styling capabilities powerful enough so that styled spreadsheets may serve as the basis for appealing custom application UIs, and so that users do not have to learn to use a separate UI builder. Furthermore, we reused the styling mechanism to implement a set of tool-generated annotations, known as the design view, that can optionally be displayed on a sheet to make the structure clearer. We provide more details about each of these topics in the remainder of this section.

2.1 Custom Layout

The "custom layout" is a powerful way for users to present their data in Espalier in a manner that feels more natural for certain forms of data. To understand the full details involved in styling, we first need to understand the structure of Espalier.
2.1.1 Split, Scope, and Span

Our model for custom structured spreadsheet layouts, known as the scoped-splits model, is designed to be general enough to support the vast majority of layout idioms we have seen while maintaining the key property that the layout updates automatically when the source data changes in size. The scoped-splits model treats rows and columns symmetrically to give users the most flexibility; it supports the nested table layout as a special case. First, we will define spans and splits, and later we will describe scopes.

A custom layout is based on a hierarchical subdivision of the height of the sheet into row spans and a hierarchical subdivision of the width of the sheet into column spans. These two hierarchies intersect to produce a two-dimensionally nested structure of rectangles, though a cell may cover more than just a leaf rectangle of this structure, which is in analogy to the "merged cell" feature of conventional spreadsheets.

Spans and Splits

An empty sheet consists of a root rectangle that is the intersection of the root row span and the root column span. A rectangle is defined to be the intersection of two spans: a row span and a column span. A row split can then be defined to break a row span into smaller row spans, producing new rectangles where the new row spans intersect existing column spans; likewise, column splits can be defined. A split is either variable or fixed. A variable split holds a set of objects of the same type (also referred to as a collection, where objects are items inside of the collection) and has one child span for each object, so that the addition or deletion of objects results in the addition or deletion of child spans. For example, the collection of a user’s friends can be stored in a variable split, where each friend is a child span of the variable split.
When the user adds a friend to the collection, an extra child span gets added to the variable split. In contrast, a fixed split has a fixed set of child spans, independent of the size of the data set. To add to the previous example of friends, each friend may have a name, a photo, or an address. Each of these fields may be a child span of the same fixed column split, and each friend is a child span for the same variable row split, resulting in a 2D table where row $i$ has all fields on friend $i$ and each column $j$ has information for field $j$ of all friends.

Figure 2-1 is an example sheet that demonstrates scopes, splits and spans. The root row span has a two-way split to split into two smaller row spans, one for Dept1 and another for Dept2. When we add a new department, we also need to add a new row span, thus this split is a variable split. The root column span has a four-way split to separate into four smaller column spans, for ProjectName, Period, Resource Charge, and Sub-total corresponding to the four yellow arrows. Because adding a new department does not add another column span for it (since every department has these four fixed attributes), this is a fixed split.

The two smaller row spans Dept1 and Dept2, each possesses a four-way fixed split defined on themselves for department title, resource charge, division name and projects corresponding to the four purple arrows. The smaller column span that corresponds to Resource Charge has a two-way variable split (DivX.1 and DivY.1) for Dept1 and three-way variable split (DivX.2, DivY.1 and DivX.1) for Dept2. Note that there is a horizontal edge dividing the cell for Resource Charge and the cells which contain the division names underneath it, but there are no edges dividing the cells in the same row as Resource Charge. This is because the cells containing ProjectName, Period and Subtotal are merged cells. Since we have the four-way fixed split defined on the row span that represents Dept1, we see that there are two purple arrows indicating the smaller row spans from that fixed split. Thus, these two smaller row spans, intersected with the column span that covers the width of the project name cell, formed two leaf rectangles which are merged in the cell.
Figure 2-1: Custom Layout

<table>
<thead>
<tr>
<th></th>
<th>Period</th>
<th>Resource Charge</th>
<th>Sub-total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DivX.1</td>
<td>DivY.1</td>
</tr>
<tr>
<td>Proj1.1</td>
<td>Jul-15</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Aug-15</td>
<td>0.7</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DivX.1</td>
<td></td>
</tr>
<tr>
<td>Proj2.1</td>
<td>Jul-15</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Aug-15</td>
<td>0.8</td>
<td>1</td>
</tr>
<tr>
<td>Proj2.2</td>
<td>Aug-15</td>
<td>0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>
Scopes

A user may want a row split to apply to only certain columns of a sheet, perhaps because vertical space of the remaining columns should be subdivided in a different way. Thus, a row split has a *scope* consisting of one or more adjacent column spans, and the child row spans that it defines are only "in scope" across those column spans. Likewise, a column split has a scope consisting of one or more adjacent row spans. For example, in Figure 2-1, under *Resource Charge*, we see that for *Dept1*, there is a two-way variable split for *DivX.1* and *DivY.1*, while for *Dept2*, there is three-way variable split for *DivX.2*, *DivY.1*, and *DivX.1*. These column splits only apply to some of the rows of the sheet.

Given that the layout needs to update automatically when the source data changes in size, the user actually builds a *layout template*, in which each variable split specifies only the associated object type and a sub-layout to be repeated for each object of the type. As the source data changes, the layout template is instantiated with the current source data to produce a *concrete layout* in which each variable split has the proper number of child spans, and this concrete layout is displayed. There is no ability to change the layout or styling for just one object of a type, other than by incorporating conditional logic into the layout template itself.

2.1.2 Allocations

Allocations in Espalier can be thought of the equivalent of cells in a conventional spreadsheet. An allocation can hold content within itself or can be empty, but all contents are covered by an allocation. Recall that a rectangle is formed by the intersection of a row span and a column span. Thus, an allocation might cover more than one rectangle.
An allocation can either be merged or non-merged. When an allocation is merged, it merges across more than one span. If it is only merged across one span, it is considered as a *non-merged allocation*. For example, in Figure 2-1, the cell with "Project Name" inside is a merged allocation, since it merges across two children spans of a fixed split (the two of the four small purple arrows). But the cells that have the division names labels are unmerged because they only cover a single span.

### 2.1.3 Why Scoped Splits

Being able to assign scopes to splits is important so splits do not have to extend across the entire page, and multiple splits can be defined on the same parent span that divide the parent span differently. This allows us to support multiple tables, nested or not, on the same spreadsheet without having to align all desired rows and columns perfectly. The custom layout provides a concise and easy method to facilitate the presentation of structured data. Some data do not naturally fit the older nested columns layout, as not every type of data is suited for a nested structure.

### 2.1.4 Default View, Design View, and Unprivileged View

The custom layout supports three views on the same underlying spreadsheet: default view, design view, and unprivileged view. Users can build an application in either the default or design view, or interchange them as needed, and display what the end user for the application will view by switching to unprivileged view at anytime. The default view option is the most similar of our views to that of conventional spreadsheets. It possesses rows and columns to showcase our scoped-splits layout. Figure 2-1 above is in default view.

The design view is a transformation from the default view, created by adding spans to make the structure more obvious. Additional details on the design view will be discussed in later chapters. The unprivileged view is what a user of the built application would see and interact with. It serves the same purpose as the default view,
except that it hides all borders unless explicitly specified, such that users of the application will not be able to know whether they are interacting with a spreadsheet or a regular web page. The supplemental figures under the sizing chapter are displayed in unprivileged view, to show what end users would see when interacting with the built application.

2.2 Espalier as an App Builder

The purpose of styling the custom layout is to make the custom layout views powerful enough for users to use it as an application builder. We want our styling capabilities to support building appealing custom application UIs so that our users do not feel the need to find another UI builder. Whether our users only want to build a sheet that represents structured data without too much complexity, or if they would rather take a step further and add formulas to use the sheet to build a custom application, Espalier can fully support these needs.

To build a custom application, an administrator can generate context-specific views by entering formulas into Espalier, just as if the administrator was simply building a sheet, where they would need to specify what goes into each view, how each component is styled, what behaviors are allowed based on the user who is accessing this view, and how these views are related. A person who is accessing the application the administrator has built can log in and view the pages that the person has access to as if it were a normal web application. These users would not be aware that the specific items on the page that he or she is viewing may be from a cell in our spreadsheet.
Chapter 3

Styling Properties

3.1 Overview

To enable styling in both design and default view, we need to decide on a set of styling properties to support. We selected these properties because they are most commonly-used when styling a website. There are two types of styling properties: ones that are associated with allocations, and others that are associated with spans. Users can specify both a subset of the allocation styles as well as a subset of spans styles while building an application in either default or design view via the styling menu. Our system sets defaults for certain styles and a subset of these styles are not changeable from the default by users.

3.2 The Effect of Design View Transformation

Figure 3-1 provides the list of styling properties associated with allocations and their restrictions. Figure 3-2 provides the list of styling properties associated with spans and their restrictions. To fully understand these property lists, we need to first understand how the design view transformation affect our styling properties. The default view only contains user-created spans and allocations, while the design view contains everything in default view as well as the any special padding spans that get added only during this transformation. Transformation from default view to design...
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Allowed Values</th>
<th>Has Default Values for User Cells</th>
<th>How Design View Utilize it</th>
<th>Changeable by Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background Color</td>
<td>Any RGB</td>
<td>White</td>
<td>Added columns/rows have special background color</td>
<td>Yes</td>
</tr>
<tr>
<td>Text Format</td>
<td>Bold, Italic, Underlined, Regular</td>
<td>References are underlined; hyperlinks are underlined</td>
<td>Labels are in Italic</td>
<td>Yes</td>
</tr>
<tr>
<td>Text Orientation</td>
<td>Horizontal, Sideways</td>
<td>Horizontal</td>
<td>Name of object type is sideways</td>
<td>No</td>
</tr>
<tr>
<td>Image Editability</td>
<td>Boolean</td>
<td>No</td>
<td>No special usage</td>
<td>No</td>
</tr>
<tr>
<td>Text Alignment</td>
<td>Left, Right, Center</td>
<td>Right for numbers, Left for non-numbers</td>
<td>No special usage</td>
<td>Yes</td>
</tr>
<tr>
<td>Text Font</td>
<td>All Valid Fonts</td>
<td>Default Font (Source Sans Pro)</td>
<td>No special usage</td>
<td>Yes</td>
</tr>
<tr>
<td>Font Size</td>
<td>All Valid Sizes</td>
<td>14</td>
<td>No Special usage</td>
<td>Yes</td>
</tr>
<tr>
<td>Text Color</td>
<td>Any RGB</td>
<td>Gray for computed values, blue for hyperlinks, black otherwise</td>
<td>No special usage</td>
<td>Yes</td>
</tr>
<tr>
<td>Border Style</td>
<td>{ Color: string, Visibility: boolean, isDouble: boolean }</td>
<td>Single visible gray border for all user cells</td>
<td>All splits are highlighted using double border rectangles. Variable splits have a special color</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 3-1: Allocation Style Properties
view does not change the values of any styling properties specified in default view. Design view transformation can only change the styling of the newly added spans. Once the transformation is done, the added spans are locked such that no further structural or stylistic modifications can be made. While in design view mode, the user can still interact with and modify the user cells as the user normally would in default view. The design-view added spans through transformation are purely padding needed to make the structure clear, and thus should not be editable by users.

3.3 Styling Properties

In Figure 3-1 and 3-2, the allowed values column specifies all possible values to which users could change the styles of a user allocation or span, if it is indicated to be "changeable by users". All styles labeled to be "changeable by users" are changeable on the user-created allocations or spans, and do not apply for the design view only spans. Design view transformation sets the background color for the cells added during the transformation to be various shades of a light green color. A later section on design view will provide more details on all design view defaults.

To enable adding images to application, Image was added as a new basic data type to Espalier. Users are able to upload photos in addition to typing data into
cells. Once a user selects a cell as of type \textit{Image}, the image edibility property is set to true and an "upload image" icon appears. When an image is uploaded, the image gets displayed in place and clicking the image again allows the user to upload a different image to replace the old image. To use the image elsewhere in the application, the user can refer to it by specifying in the formula field of the structuring menu and the image editability is set to false by our system because the allocation it is in is bind by the formula and is not editable. To change the images, users would need to go back to the sheet where the original images are uploaded and re-upload there because the images still have image editability set to true.

Border style is slightly more complicated. Each of the four borders around an allocation possesses an associated border style, allowing each of the four borders to be styled separately. All borders are single and gray by default. Our system during the design view sets certain borders to be double and gives them a special color during the transformation. Again, more details on design view styling will be explained in later sections.

Minimum Span Size is set by our system during the design view transformation on the added spans. Since they are empty, without a proper minimum size, they may get sized down to zero. Users cannot set the minimum span size property – this is intended for design view only.

Users can further set sizing of a span to any allowed values using the styling menu. The sizing chapter later will describe the entire process of sizing and what those allowed values mean.

In summary, there are only two ways styles can be set – either users set them or the system sets them. Whenever the user wants to set the styling on any user cells or spans, the user would need to do so by using the styling menu. The system sets default styles upon creation, and sets styling on design view spans during the trans-
formation which creates them. The next two sections will be focus on these two ways of how styles are being set.

### 3.4 The Styling Menu

Any styling setting and modifications by users are done using the styling menu. The entire menu is composed of two parts, that of allocation styling part and of span styling. The allocation part of the styling menu allows user to specify the parameters listed in Figure 3-1. The span part of the styling menu will be described in the sizing chapter along with the algorithms we used for sizing and how our choice of sizing parameters affected our design for the menu.

Figure 3-3 is the allocation sub-menu. It is subdivided into components that handle text, background, and border styling. To use the menu, users can select any cell by clicking inside, or drag and select a rectangle that covers multiple cells. Our selection determination code will identify the allocation that covers the selection. The components in our styling menu will always display the style of the current allocation selected. We designed the allocation sub-menu to resemble the familiar interface of Microsoft Word and Excel in order to reduce users’ cognitive load while familiarizing with our styling menu.

The text column of the allocation sub-menu allows users to specify the text format of the current selection. Each text format can be selected and unselected independently. More than one text format could be selected, and when all are unselected, this is equivalent to regular text formatting. When selected, the background darkens.
Text alignment works similarly to text formatting, except only one alignment can be chosen at one time. This alignment option does not only apply to text, but also applies to other datatypes inside, such as hyperlinks and buttons. The font, color, and size selector are also very similar to the equivalent in Word and Excel (Figure 3-4a and Figure 3-4b). Note that in Figure 3-3, the font selector has no font displayed when the page is loaded initially because no selections have been made. Our system default font shows up as "System Default Font" in our font selector.

The border column of the allocation sub-menu is composed of a border selector on top, where users can select a single edge. The selector highlights the edge, and the color and visibility of the edge are displayed below of the current selected allocation.
To change the style of an edge, user would select the edge using the selector, and modify its color and visibility below (Figure 3-5).

### 3.5 Store Example with Styling Properties

We built a multipage store application to demonstrate how to use our styles when building a real application (Figure 3-6). The color palette used for this application is composed of a brick red, mustard yellow, dark blue, and pastel blue color. The font choice was Tahoma.

Figure 3-6a is the login page of our store. The welcome message is built by putting a formula on the cell that is the string "Welcome to the Espalier store!". By default, the specified string is gray. We changed its color to brick red and set its size to 30 px. We further made the username prompt mustard yellow and italicized it. The usernames come from a collection of users, so we simply needed to set the background color to pastel blue and text color to dark blue on one of them, and this style gets replicated to all users. This is not a property of styling but rather of structure. The style always gets set on the template allocation, which always gets replicated the number of items in a collection times during the layout instantiation.

Figure 3-6b is the home page of our store. Starting from the home page, all fol-
lowing pages all have the same menu bar on top. We built the menu bar by using a hyperlink to the corresponding page in all menu options. By default, hyperlinks are blue and underlined. For our purposes, we altered it to be white and with no underline. We then italicized the login username. All hyperlink and texts are aligned left by default, so we right aligned the menu options. To emphasize the search bar, we put a mustard yellow color on each side of the border of the cell dedicated as the input. The images for grocery products are referenced from a master data sheet which we uploaded and stored all data relevant to this application. Thus, these images are uneditable.

Figure 3-6c displays the product page of our store. In our home page, when a user clicks on a product, the user would be taken to this page. Here, we left the numbers to have their default right alignment to clearly separate them from their labels. We highlighted the quantity entry by bolding the text and changing the background color to our pastel blue color choice.

Finally, Figure 3-6d demonstrates the cart page of our store. In our product page, when a user clicks the “add to cart” button, the user is taken to this page. Each column of this table is a child span of a fixed split. To put a mustard yellow border around each entry, we set all edges to the mustard yellow color and hid the corresponding edges for each column. To highlight the quantity column, we made all borders visible and dark blue. Since numbers are right-aligned by default, and buttons are left-aligned, the buttons would be very close to the prices by default. To ameliorate this concern, we overrode the default left-alignment of buttons and set these buttons to right-aligned instead.
Welcome to the Espalier store!

Choose your username:
Alice
Bob
Carol

(a) Login Page

(b) Home Page
(c) Product Page

(d) Cart Page

Figure 3-6: Store Application
Chapter 4

Design View Styling

4.1 Design View Transformation

The design view transformation begins with a layout template. The layout template contains information about the overall structure of the data, such as information including types of splits, children spans and their scopes, and collection names for variable splits. Every item in the same collection will have the same number of rows and columns added in design view, which allows spaces to draw boundaries that group these items together.

The design view is implemented as a transformation that evaluates the user’s layout template and generates a new temporary layout template that contains information about which row or column spans need to be added for each object type and what should appear in them, such as the name of each object type.

Next, we instantiate the design view by utilizing the aforementioned temporary template that has information on what the design view needs to display for each collection, and make an equivalent number of copies to the number of objects of in that collection and renders it. This is the same instantiation process that is applied to the user’s original layout template in the default view. Transforming to design view adds new row spans and column spans to the sheet and styles them. Recall that it does not
change the style of user cells and user spans, which are created by users that are already in default view before the transformation.

4.2 Design View Styling

As part of the transformation process, each one of the added spans and allocations receive their fixed style. These styles are added solely for the purpose of design view spans and are, therefore, not modifiable by users. In the below sections, we describe the styles of those design view added spans and how they are set in our system.

4.2.1 Drawing Boxes Around Each Collection

To distinguish between the two different types of splits, as well as indicating the scope of those children spans, we put a red box around each child span of a variable split, and gray box around each child span of a fixed split. The box always has two double edges on the axis of the span, and two singles edges on the other axis. The scope of the span is indicated by the range of the two double edges. To distinguish the design view spans from the regular user-added spans, we provide the design view spans with a shade of light green.

In Figure 4-1, we can clearly see that every user of the user entry collection has a red box around it, and that the scope of each child is also the scope of the variable split. Furthermore, we note that this entire collection is enclosed in a gray box, which is one of the child spans of the row split defined on the root row span. For the root column span, we observe that there is a fixed split defined on it and that the scope goes all the way across the page. In this case, the scope of each child span of this fixed split is the root row span. The user spans are four of the six children of the fixed split. The other two were added to compensate for space to draw those double borders for the user entry collection on either end.

The behavior mentioned above is implemented by defining an empty allocation to
cover the entire box area and depending on whether it comes from a fixed or variable
split, as well as which axis the split is defined on, we add the appropriate border style
to these allocations. Since the rendering part of our application draws allocations one
by one from left to right and then top to bottom, the later drawn allocations would
thus cover the edge of any previously draw allocations which they share an edge with.
To make sure that in the design view that the design view boxes do not acciden-
tally cover the user allocation’s edges, we assign the z-index of the these allocations
a negative value. We make certain that regular user cells always have a non-negative
z-index so that regardless of the drawing order, these boxes cannot accidentally cover
the border of the user’s allocations, which would give a false appearance of the actual
defined styles underneath.

Additionally, to aid in both visualizing the depth of the splits and ensuring that
splits defined inside of splits are clear to the users, the design view added spans utilize
three different shades of the same light green color. The color is determined by the
color at the index computed using a formula of the \( \text{depth} \mod 3 \), where \( \text{depth} \) is the
depth of the collection. We decided on three shades as three is the most number of
shades of the same color that are different enough for users to tell a difference, while
not too different so that our background colors distract users from focusing on the
boxes.

4.2.2 Collection Names

In addition to adjusting background color and drawing appropriate boxes, we also
label every collection by their collection name right next to the double borders of
the variable splits’ children. We prefer the collection names to be written in the
direction that the collection spans across, right above the double borders as shown
for the user entry collection in Figure 4-1. This is simple to implement for horizontal
collections. Similarly to drawing the boxes, we start by creating an allocation that
covers an area equivalent to that of the blue cell’s area in Figure 4-2 for the horizontal
collection, then add the collection name, center align and italicize. This process is simple because the text, by default, shows up in the direction needed. However, for vertical collections, the desired behavior as shown in Figure 4-2 proves to be more difficult, as there are no existing CSS styles or combinations of them to align the text in such way. Recall that text orientation is one of the parameters we chose support for allocation styles. We created this parameter to handle this specific case of writing sideways text next to vertical collections. When the renderer is drawing rectangles, it will normally draw a single "div" element for that rectangle. However, in this case where the allocation style is set to sideways text, it will draw another div inside of this parent div that it normally draws and place the text which is center aligned and italicized inside the inner div. This inner div has its width and height switched from that of its parent, and is translated to an adjusted location, so that when it is transformed by 90 degrees, it perfectly overlaps with the parent rectangle. Since the text rotates with the inner div, it ends up in the correct location for vertical
collections.

4.2.3 Field Names

Finally, fields names, by default, are blank and must be specified before users can refer to them in a formula. To help users keep track the label fields, the design view transformation adds another span right above each user span to hold the field name. To style these field names, we set the background to a blueish green color, and the text to a smaller size as well as setting a small minimum size for the spans dedicated for field names.
Chapter 5

Sizing

5.1 Introduction

5.1.1 Old Sizing Algorithm

A simple sizing algorithm existed in the previous version of Espalier, which we will refer to as the old sizing algorithm. This old algorithm simply traverses the sheet and assigns each span a size to fit the content inside. To compute the sizes of the spans, the old algorithm measures the rendered size of all contents it visits and assigns the size of a parent span to the sum of sizes of its children spans. Figure 5-1 shows the case that when multiple splits exist within a single span and each split has a different sum, the old algorithm assigns the parent span (highlighted blue in the figure) the maximum size of all splits inside (the size of split2). Next, the old algorithm adds an extra space span after the last span of every other split so its new size is equal to the size of the largest split. The striped regions indicate the extra space spans added to ensure that split1 and split3 match the size of split2. Based on computed sizes for all spans, the offset computation assigns an absolute positioning and size of each rectangle to draw the spreadsheet. In the case of empty spans, instead of having zero size, a default minimum size is assigned so users can still edit and select the span.
5.1.2 Motivations for the New Sizing Algorithm

Figure 5-1 shows that the sheet in the old sizing algorithm is always as big as its content needs. This makes the pages unresponsive to each unique user’s window size. A full page that was built on one user’s window may only be displayed as a half page on another user’s window whose size is twice as big due to the absolute positioning of each rectangle in the spreadsheet. No constraint can be specified to each span when window size changes. We designed a new algorithm which allows users to specify three basic parameters (content, flex, and exact) under a set of settings to indicate how the sizes of spans should change according to window size while preserving the padding of extra spaces and having a minimum default size that the old sizing algorithm used.
5.2 Sizing Parameters

5.2.1 Parameters and Their Combinations

The new sizing algorithm, detailed in section 5.3, computes and assigns sizes to spans using the values of the three basic parameters below. Users can assign values to a subset of these parameters via the styling menu based on the sizing setting they choose. More details follow in the combinations section below. Under these definitions, the old sizing algorithm would be equivalent to having a setting of ContentOnly with the associated values set to Content: True, Flex: 0, Exact: 0 on all spans, including the root row and column spans.

1. **Content**. A Boolean indicating whether the span size should fit its content.

2. **Flex**. A positive integer to indicate how much the sizing of the span should shrink or expand when the size of the window changes. A zero indicates the span is not flexible and does not respond to changes in window sizes. Any non-zero positive integer indicates a certain portion of the extra space (if any) that should be added to the size of this span. Specifically, the portion of the extra space the span takes is computed by its assigned flex value, divided by total assigned flex value of all children of the parent split.

3. **Exact**. A positive integer indicates the exact number of pixels the span’s size should be set as. A zero indicates that the size of a span does not have to be exact (to stay consistent with other parameters with flex=0 meaning non-flex), rather than indicating that it has an exact size of zero. If a user want a span to have zero size, then the user can just delete the span. Setting a span to be size zero would make it unselectable and unusable.

Not all possible combinations of the above three basic sizing parameters have logical meanings. We chose to support a subset that do, and grouped them under different categories, which we call *settings*. The system stores the values of each of the three basic parameters and the setting their combination is under. Each supported
<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Definition</th>
<th>Default Combination</th>
<th>Changeable After Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContentOnly</td>
<td>Size of the span should solely be based on the contents inside</td>
<td>{content: true, flex: 0, exact: 0}</td>
<td>No values of the three basic sizing parameters can be changed</td>
<td></td>
</tr>
<tr>
<td>FlexOnly</td>
<td>Size of the span should only be based on the extra space of the parent split</td>
<td>{content: false, flex: 1, exact: 0}</td>
<td>Only values for <code>flex</code> can be changed</td>
<td></td>
</tr>
<tr>
<td>ExactOnly</td>
<td>Size of the span is fixed to be a certain number of pixels that a user specifies</td>
<td>{content: false, flex: 0, exact: 48}^{0}</td>
<td>Only values for <code>exact</code> can be changed</td>
<td></td>
</tr>
<tr>
<td>MaxOfExactAndContent</td>
<td>Size of the span should be max (exact value, size of span’s content)</td>
<td>{content: true, flex: 0, exact: 48}^{0}</td>
<td>Only values for <code>exact</code> can be changed</td>
<td></td>
</tr>
<tr>
<td>ContentPlusFlex</td>
<td>Size of the span is the sum (size of the contents inside, any portion of extra space indicated by the flex value)</td>
<td>{content: true, flex: 1, exact: 0}</td>
<td>Only values for <code>flex</code> can be changed</td>
<td></td>
</tr>
<tr>
<td>ExactPlusFlex</td>
<td>Size of the span is sum(exact value specified, any portion of extra space indicated by the flex value)</td>
<td>{content: false, flex: 1, specific: 48}</td>
<td>Both values for <code>exact</code> and <code>flex</code> are changeable</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Settings, Defaults, and Constraints

Each setting has a default combination of the three parameters, which we set for the users when they select a setting. This way, users are not required to fill in specific values for those parameters and can just leave them as defaults if they are satisfied. In addition to defaults, every setting comes with a set of constraints on which of the three basic sizing parameters can be changed to stay in the setting. Table 5.1 lists the supported settings, their defaults, and their restrictions.

5.2.2 Sizing in the Styling Menu

Sizing options occupy the last section of the styling menu. Users directly click on the spreadsheet to make a selection. Once a selection is made, similarly to other styling options, the current setting and values of sizing parameters for the selection’s span on the currently selected axis are displayed and can be modified in place (Figure 5-2).

When the user clicks a cell or clicks and drags to select a region covering multiple

---

^{0}The number 48 is chosen because it is the default minimum size for a column span
Figure 5-2: Sizing section of the styling menu: (1) Axis selector. By default, it is set to row. (2) Setting selector. Displays the current setting of the selected cell’s span on the axis indicated by the axis selector and allows user to change between different settings. (3) Combinations modifier. Displays the current values of each of the three basic sizing parameters, and allows users to change values of eligible parameters. Here we can see that all three parameters are disabled because user has chosen a setting of "Fit Content" which sets content to true, flexible to zero, exact to zero, and does not allow any of these values to be modified.

cells, the system gets the merged rectangle covering this selection, and this rectangle is highlighted on the spreadsheet. Figure 5-1 shows the rectangle corresponding to the parent span of the three splits which the user selected. We then get the merged span of the allocation, which covers this selected rectangle on the current axis. This merged span should be ideally a single span (the merged span merges over just a single span) which we assume the user intended to get and set the style for. However, if the allocation is merged over multiple spans on the current axis, this merged span would have a first and last span which are not the same. We cannot identify which span the user intend to set the style for, so we disable any modifications to the span’s sizing by disabling the setting selector and the combinations modifier (Figure 5-3).

To adjust the span’s sizing constraint, the user would first use the axis selector to choose between row and column as the current axis, as in looking up the current style. Secondly, the user uses the setting selector to select among the six supported settings which modify the values of the three basic sizing parameters to each setting’s default configuration and disable any sizing parameters that are unchangeable under this setting. The user can further modify the values for any non-disabled parameter.
Figure 5-3: Current selection is merged over multiple spans on the column axis, thus there is not a single span whose style should be displayed and allowed to be modified. By default the sizing portion in the styling menu displays the default setting of ContentOnly and disables both the setting and the sizing parameters to indicate there is no sizing on the selected span that can be displayed or modified.

5.3 The sizing algorithm

Whenever users specify a constraint for a span, the sizing algorithm is rerun to compute the sizes of each span and the entire spreadsheet re-renders to reflect the updated sizes.

Though the entire sizing algorithm may be written as one unit and as a single round of computation, we have divided it into two rounds to support incrementalization: the bottom-up part, which computes the sizes of contents inside each span and any adjustments needed, and the top-down part which assigns each span a size. This way, when users change the sizing of a span, we only need to recompute the sizing for the affected spans because the top-down and bottom-up part can be run separately. In order to make the bottom-up and top-down algorithms used in each part more clear, we semantically define the helper sizes computed in each part. Table 5.2 details the helper sizes of a span and which round of computation in which each size is computed.
Below are the two algorithms used (Algorithm 1 and Algorithm 2). The \texttt{computeSizeBottomUp} function is called on each parent span and computes the outer and inner default sizes of the parent span by looking up the previously computed sizes of its children spans and adjusting as needed. The outer default sizes for children spans are computed at the child level and adjusted at the parent level to fit merged allocations. The \texttt{assignSizeTopDown} function is called on each parent span and looks up the computed size for the parent span to assign sizes for the related children spans. For more details on these algorithms, refer to section 5.3.3 and 5.3.4.

The overall process of determining sizes for each span, given the sizing properties, executes twice, once for each axis, and for two rounds (bottom up and top down) each time. We use an implemented axis-traverser to traverse the sheet, call \texttt{computeSizeBottomUp} on all spans, and call \texttt{assignSizeTopDown} on all spans. The axis-traverser ensures two things:

1. \texttt{computeSizeBottomUp} is called for all children spans before the parent span itself. Thus, the sizes of the children are all computed when \texttt{computeSizeBottomUp} is called on the parent and looks up the computed sizes for the children.

2. \texttt{assignSizeTopDown} is called for a parent span before its children spans. Thus, the sizes of the children spans are all assigned when \texttt{assignSizeTopDown} is called on each child and looks up the assigned sizes.

\subsection{A Sizing Example}

Figure 5-4 shows an example sheet after running the sizing algorithm with specified sizing constraints on each row span. This example will be used to demonstrate only

\footnote{To distinguish the outer default size from the outer actual size of a span, think of the outer default size as the minimum size this span need to be under its sizing constraint, while the outer actual size of the span is made by add any padding necessary in the case of extra space left over to the outer default size.}
Table 5.2: Helper Sizes Definition

<table>
<thead>
<tr>
<th>Size</th>
<th>Definition</th>
<th>Computed During</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer Default Size&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Default size the span should be. If a certain size is needed, it’s that specific size. Else, it’s the inner default size.</td>
<td>bottom-up</td>
</tr>
<tr>
<td>Inner Default Size</td>
<td>Size needed to fit the contents inside the span</td>
<td>bottom-up</td>
</tr>
<tr>
<td>Outer Actual Size&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Actual size the span should be set at (This is the resulting size rendering uses to draw)</td>
<td>top-down</td>
</tr>
<tr>
<td>Inner Actual Size</td>
<td>Actual size of the contents inside of the span (This is used to determine whether or not a scroll bar is needed to be placed inside of the span)</td>
<td>top-down</td>
</tr>
</tbody>
</table>

how the row span sizes are computed and assigned. The next two sections will walk through this example as we step through the sizing algorithm.

The root row span in our example sheet is divided by a fixed split into two spans, $S_1$ and $S_2$, indicated by the orange arrows. Both of their scopes cover the entire root column span. $S_1$ is further divided using three different splits. $Split1$ has two children spans, indicated by the blue arrows, and their scopes extend from the blue arrows to the red arrows. Similarly, $split2$’s children have their scope from the red arrows to the green arrows and $split3$’s children have their scope from the green arrows to the end of the page on the right. $Split2$ and $split3$ hold the same content and structure, except child1 of $split3$ has a FlexOnly setting with a flex value of 1. Both children of $split1$ has FlexOnly settings. The first child has flex value of 2, while the second child has flex value of 1. All others spans have a setting of ContentOnly.
5.3.2 Algorithm Setup

Both computeSizeBottomUp and assignSizeTopDown are called on a given span. In our internal representation, each span is uniquely identified by its ID. During the two rounds of computation, we keep track of the currently computed helper sizes (Table 5.2) so they can be looked up later during subsequent calls to computeSizeBottomUp and assignSizeTopDown. To do so, we keep four maps, one for each helper size, which have the helper sizes mapped to the span IDs. Each map is initially empty, and fills up as the axis-traverser calls computeSizeBottomUp and assignSizeTopDown during which the helper sizes get computed. Note that while traversing, the axis-traverser also records a list of allocations (represented by their unique IDs) and a list of splits (represented by their unique IDs) under each span.

5.3.3 Bottom-up Part of the Sizing Algorithm

The computeSizeBottomUp function computes the minimum size (the outer default size) required to fit the content for all spans on which it is called. It does so by first adjusting the computed minimum sizes for each child span to fit any merged allocations, and summing up the child span sizes to get the inner default size. Finally, based on the setting of the span and the values of the parameters, it outputs the
Sizes for Children Spans in Example Sheet

In our example, since the children spans of the splits 1, 2, and 3 are the leaf spans (they do not have any splits inside), they would be the first spans on which `computeSizeBottomUp` gets called. Since there is only a single unmerged allocation inside each of them (the allocation which has their name written inside), lines 4-13 are skipped and its rendered height, which we measure to be 20 px, becomes the `maxNonMergedAllocContentSize`. Since lines 17-20 are also skipped, the `maxSplitTotalODSize` is zero and `innerDefaultSize` becomes 20 px for all children spans of splits. All children from `split2` have sizing set to `ContentOnly` so all of their `outerDefaultSizes` are assigned to 20 px. This is the same case for children from `split3`, except that `child1` has `FlexOnly` setting. Thus, its outer default size is set to 0. Similarly, in `split1` both children have their outer default size set to 0. Table 5.3 summarizes these values. After the children sizes have been computed, we now proceed to compute sizes for $S_1$ and $S_2$.

Sorting Allocations inside of $S_1$

For $S_1$, the function looks up the list of allocations under its ID in the appropriate map and sorts the allocations that come from the same split in the order of the last
span each allocation merges across. Figure 5-5 has all allocations annotated with their labels in our sizing example. We can see that in Figure 5-5, allocations with the building image in \textit{split}2 is labeled as \textit{B1} and in \textit{split}3 as \textit{B2}. The allocations with the frosted tree image in \textit{split}2 and \textit{split}3 are labeled \textit{F1} and \textit{F2}, respectively. Furthermore, the allocation which contains the string "text allocation" is labeled as \textit{text allocation}, and the allocation which covers the lilac image is labeled as \textit{flower allocation}. Sorting would need to ensure that \textit{B1} comes before \textit{F1} because the last span that \textit{B1} merges across (\textit{split}2,\textit{child}2) comes before the last span \textit{F1} merges across (\textit{split}2,\textit{child}3). Similarly, \textit{B2} needs to come before \textit{F2}. The order of others allocations in our sorted list does not matter. For our example, the order of allocations is \([\textit{B1}, \textit{F1}, \textit{B2}, \textit{F2}, \textit{text allocation}, \textit{flower allocation}]\).

\section*{Why Sorting}

This sorting behavior is important for the size-adjusting section of the algorithm to minimize the amount of padding needed to fit the merged allocations in an efficient way. The idea to minimize the padding is based on the fact that when two merged allocations have any overlaps, the minimum padding strategy is to pad in the overlap area, ensuring that both allocations receive the padding. Overlaps can happen only in merged allocations within the same split. Thus, when those allocations are sorted by
the order of their last child, the last child which receives the padding is guaranteed
to be a part of the overlap if any exists. Sorting allocations by their last span is
relatively non-costly as it needs to only look up the index of the last span in the
subspans of the parent split.

Adjusting Children’s Outer Default Sizes for S1

Revisiting our example sheet in Figure 5-4, both of the smaller images are 100 px by
100 px, making allocations $B_1$, $F_1$, $B_2$, $F_2$ too big to fit on the spans they merge
across. We now look through our sorted allocation list. We process $B_1$ first, looking
up its children’s outer default sizes in the appropriate map (Table 5.3). $B_1$ needs an
extra $(100 - 20 - 20) = 60$ px of space, and since both of the child spans it merges
across are non-flex, by default, all 60 px get padded to the outer default size of last
span ($split_2, child_2$). We apply this padding by adding 60 px to the outer default size
of $split_2, child_2$ in our map.

Now, we process $F_1$. We sum the children spans it merges across by looking up
the children’s outer default sizes in the appropriate map (note that $split_2, child_2$ has
an updated size of 80 px now), and realize that it fits perfectly. No sizing adjustment
is needed.

Next, we process $B_2$. Since $split_3, child_1$ is FlexOnly and thus has outer default size
of 0, $B_2$ will need an extra $(100 - 20) = 80$ px of space. Because we always attempt to
pad extra space to the size of any flex spans before resorting to the last span, all of 80
px of padding get added to the outer default size of $split_3, child_1$. Since it used to have
an outer default size of 0, now its updated size becomes 80 px. If $split_3, child_2$ were
also flex, then it would get its share of the 80 px extra space according to its flex value.

The final merged allocation we process is $F_2$. Similar to $B_1$, its children spans’
outer default sizes sum up to be $(20 + 20) = 40$ and it needs 60 px of extra space
which gets padded to the last span it merges across – $split_3, child_3$. Table 5.4 shows
Table 5.4: Helper Sizes After Adjusting for Merged Allocations

<table>
<thead>
<tr>
<th>Span</th>
<th>Sizing Constraint</th>
<th>Outer Default Sizes</th>
<th>Inner Default Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>split1, child1</td>
<td>FlexOnly with flex value = 2</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>split1, child2</td>
<td>FlexOnly with flex value = 1</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>split2, child1</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>split2, child2</td>
<td>ContentOnly</td>
<td>20+60 = 80</td>
<td>20</td>
</tr>
<tr>
<td>split2, child3-5</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>split3, child1</td>
<td>FlexOnly with flex value = 1</td>
<td>0+80 = 80</td>
<td>20</td>
</tr>
<tr>
<td>split3, child2</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>split3, child3</td>
<td>ContentOnly</td>
<td>20+60 = 80</td>
<td>20</td>
</tr>
<tr>
<td>split3, child4-5</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

the updated helper sizes after adjusting for merged-allocations.

**Finding maxNonMergedAllocContentSize for S1**

We now look at the last two allocations in our sorted list. Both are unmerged allocations. Specifically, *text allocation* is measured to have a rendered height of 20 px, which is less than the rendered height of *flower allocation*, 200 px. Thus, *maxNonMergedAllocContentSize* is 200 px.

**Finding maxSplitTotalODSize and innerDefaultSize for S1**

After finishing the allocations list, we are at line 17, where we begin to sum up the updated outer default sizes, found in the appropriate map, of all children spans in each split. In Table 5.4, we can see that *split3* has the maximum total outer default size of 80 + 20 + 80 + 20 + 20 = 220 px. This is larger than the 200 px of *maxNonMergedAllocContentSize*, so *innerDefaultSize* of S1 gets assigned to 220 px.

Once we have this inner default size, we use it to decide the outer default size of our parent span. This outer default size can be thought of as the minimum size that
our parent span can be, provided its own specified sizing constraint. Thus, we assign
the outer default size to our parent span based on which sizes should be the minimum
size under the sizing setting to which parent span is assigned (Lines 21-30). In this
case, $S_1$ is set to ContentOnly, so its outer default size is set to its inner default size
of 220 px and added to our appropriate map.

**Bottom-up for $S_2$**

Next, computeSizeBottomUp is called on $S_2$. It has an empty list of allocations as
well as splits. Thus, lines 1-20 are skipped and its inner default size is set to 0. Since
$S_2$ has default setting of ContentOnly, its outer default size is set to 0 as well. This is
cased by $S_2$ not having any content inside. Line 31 catches this case and adjusts this
outer default size to be a system’s constant minimum required size (in our example, we
used 24). The purpose of this line is to ensure that empty spans which have recently
been created do not get assigned zero size and thus become unselectable and unusable.

**Bottom-up for Root Row Span**

Lastly, computeSizeBottomUp is called on the root row span. There are no alloca-
tions directly inside. It has a single split, dividing it into $S_1$ and $S_2$, making the
$maxSplitTotalODSize$ equal to 244. Note that since the outer default sizes of the
root spans are never used in the top-down round, these sizes do not necessarily need
to be computed. Here, we simply choose to compute the outer default size for the
root row span has ContentOnly, which gives us 244 px. Table 5.5 shows all helper
sizes computed and stored in our maps at the end of the bottom-up round.

**5.3.4 Top-down Part of the Sizing Algorithm**

The function assignSizeTopDown focuses on assigning the sizes of each children of the
current span. It does so by adding any necessary padding to each child span if there
exists extra spaces based on the window size and on the previously-assigned outer
Algorithm 1 computeSizeBottomUp(span)

1: sortedAllocations = allocations inside of span within the same split sorted by
the order of the allocation’s last span
2: for each allocation in sortedAllocations in the order they are sorted do
3:   thisAllocSize = rendered size of allocation
4:   if allocation is merged then
5:     subSpans = list of subsprans that allocation merges across
6:     totalSubSpanSizes = sum over outer default sizes of all subSpans
7:     paddingAmountNeeded = thisAllocSize – totalSubSpanSizes
8:     if paddingAmountNeeded > 0 then
9:       divide paddingAmountNeeded among the flex spans in subspans according
to their flex weights and add the amount to their outer default sizes
10:      if there are no flex spans, then add all of paddingAmountNeeded to the
outer default size of last span in subspans
11:      update any adjusted outer default size of subspans
12:   end if
13: else
14:   keep track of the maximum thisAllocSize for non-merged allocations
    as maxNonMergedAllocContentSize
15: end if
16: end for
17: for each split in splits inside of span do
18:   totalOuterDefaultSize = sum of outer default sizes for children spans inside
   of split
19:   keep track of the maximum totalOuterDefaultSize for splits inside of span
   as maxSplitTotalODSize
20: end for
21: innerDefaultSize = max(maxNonMergedAllocContentSize, maxSplitTotalODSize)
22: if sizing on span is set to MaxOfExactAndContent then
23:   outerDefaultSize = max(sizingOption.exact, innerDefaultSize)
24: else if sizing on span is set to FlexOnly then
25:   outerDefaultSize = 0
26: else if sizing on span either ContentOnly or ContentPlusFlex then
27:   outerDefaultSize = innerDefaultSize
28: else if sizing on span ExactOnly or ExactPlusFlex then
29:   outerDefaultSize = sizingOption.exact
30: end if
31: if span is empty and has sizing setting ContentOnly, then make sure its
   outerDefaultSize >= minimum required size
32: Save innerDefaultSize, outerDefaultSize
actual size of the parent (Algorithm 2). This is the part of the algorithm responsible for making the sizes of flex spans responsive to the available space.

Before `assignSizeTopDown` is called, in order to make sure that the sheet occupies the entire window, the root spans of the sheet are assigned an `outerActualSize` corresponding to the window sizes. For the purpose of our example, assume that the root row span is assigned 400 px.

**Top-down for Root Row Span**

Since the root row span in our example is the highest span, it is called first by `assignSizeTopDown`. We look up in the maps for its computed inner default size (244) and assigned outer actual size (400), and assign `innerActualSize` to 400. In this case, we have more space than its minimum size for its content, so no scroll bar is needed, as determined by lines 3-5, and the amount of padding we need will be positive. Starting from line 6, we go through each split in our list which the axis-traverser stored earlier. Here we will only find one split. Line 6-21 assigns the outer actual size of its children by adding any extra space needed to the computed outer default size for the children. The sum of the outer default sizes of its children for the root row span is $220 + 24 = 244$ and the amount of extra space as indicated by line 9,
Table 5.6: Helper Sizes After Assigning Sizes after Processing Root Row Span. Recall that the inner default sizes already computed during bottom-up. They are included here for clarity.

<table>
<thead>
<tr>
<th>Span</th>
<th>Sizing Constraint</th>
<th>Outer Actual Size</th>
<th>Inner Actual Size</th>
<th>Scrolling Needed</th>
<th>Inner Default Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Row Span</td>
<td>ContentOnly; Not Modifiable by users</td>
<td>400</td>
<td>400</td>
<td>False</td>
<td>244</td>
</tr>
<tr>
<td>S1</td>
<td>ContentOnly</td>
<td>220</td>
<td>220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>ContentOnly</td>
<td>180</td>
<td></td>
<td></td>
<td>220</td>
</tr>
</tbody>
</table>

is \(400 - 244 = 156\). The outer actual size of \(S2\) is then \(24 + 156 = 180\), and the outer actual size of \(S1\) is its outer default size of 220. Table 5.6 shows the helper sizes and their values stored in our maps at the end of this step.

**Top-down for S1, Split1**

For the purpose of our example, \(S1\) is called first. The inner default size is equal to the outer actual size (220), thus the inner actual size is assigned to 220 and no scroll bar is needed. Line 6 iterates through a list of splits inside of \(S1\). The goal of lines 6-23 is to add any extra space needed to the children spans of each split, such that when summed up, each split occupies the entire space assigned for its parent. Similar to what was done in `computeSizeBottomUp`, the function will attempt to divide the extra space among any flex children and resort to the last child if none of the children are flex. The function looks at `split1` first. The sum of outer default children sizes of `split1` is zero. All of the 220 extra space is divided into the first and second child. According to definition of flex values, the first child gets twice the amount of padding and has an outer actual size of \(0 + 220 \times \frac{2}{3} = 147\) px, while the second child will get \(0 + 220 \times \frac{1}{3} = 73\) px. Table 5.7 shows the helper sizes after `split1` is processed.

**S1, Split2**

In contrast with `split1`, the children of `split2` are all set to ContentOnly. The sum of the outer default sizes is \(80 + 4 \times 20 = 160\), which leaves us \(220 - 160 = 60\) px in extra
space. By default, all of this extra space gets added to \textit{split2, child5}, making its outer actual size $20 + 60 = 80$. For all other children, the outer actual size is assigned to its outer default size. Table 5.8 shows the helper sizes after \textit{split2} is processed.

\textbf{S1, Split3}

In contrast with \textit{split2}, the first child of \textit{split3} is assigned FlexOnly. The sum of the outer default sizes is $2 \times 80 + 3 \times 20 = 220$, which leaves us 0 px in extra space. In this case, all children’s outer actual sizes are assigned to their outer default sizes. Table 5.9 shows the helper sizes after \textit{split3} is processed.
Table 5.9: Helper Sizes after *Split3* is Processed

<table>
<thead>
<tr>
<th>Span</th>
<th>Sizing Constraint</th>
<th>Outer Actual Size</th>
<th>Inner Actual Size</th>
<th>Scrolling Needed</th>
<th>Inner Default Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Row Span</td>
<td>ContentOnly; Not Modifiable by users</td>
<td>400</td>
<td>400</td>
<td>False</td>
<td>244</td>
</tr>
<tr>
<td>S1</td>
<td>ContentOnly</td>
<td>220</td>
<td>220</td>
<td>False</td>
<td>220</td>
</tr>
<tr>
<td>split1, child1</td>
<td>FlexOnly with flex value = 2</td>
<td>220*2/3 = 147</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split1, child2</td>
<td>FlexOnly with flex value = 1</td>
<td>220*1/3 = 73</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split2, child1</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split2, child2</td>
<td>ContentOnly</td>
<td>80</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split2, child3-4</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split2, child5</td>
<td>ContentOnly</td>
<td>20 + 60 = 80</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split3, child1</td>
<td>FlexOnly with flex value = 1</td>
<td>80</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split3, child2</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split3, child3</td>
<td>ContentOnly</td>
<td>80</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split3, child4-5</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**S2 and Other Children Spans**

The inner actual size for *S2* is \(\max(0, 180) = 180\) and no scrolling is needed. There are no splits inside of *S2*, so it does not need to assign any more children sizes. Similarly to all other children spans of *split1*, *2*, and *3*, no scrolling is needed on these children spans and they do not contain further splits. Table 5.10 is the final result after running the top-down round. The outer actual sizes are the sizes we ultimately use in rendering our example sheet (Figure 5-4).

### 5.4 The Store Example with Sizing

We have augmented the aforementioned store application example to include sizing constraints on the spans in order to demonstrate basic sizing behavior that such a website might desire. Recall the store example in the Styling Properties chapter and the different pages inside. In the "Styling Properties" chapter, we detailed how we specified the allocations styles on the application. In this section, we detail how we will continue with the example and provide details on the sizing of the spans in the store application.

To ensure that our pages maintain a consistent style, we wanted to make the main
Table 5.10: Helper Sizes after all Spans are Processed

<table>
<thead>
<tr>
<th>Span</th>
<th>Sizing Constraint</th>
<th>Outer Actual Size</th>
<th>Inner Actual Size</th>
<th>Scrolling Needed</th>
<th>Inner Default Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Row Span</td>
<td>ContentOnly; Not Modifiable by users</td>
<td>400</td>
<td>400</td>
<td>False</td>
<td>244</td>
</tr>
<tr>
<td>S1</td>
<td>ContentOnly</td>
<td>220</td>
<td>220</td>
<td>False</td>
<td>220</td>
</tr>
<tr>
<td>S2</td>
<td>ContentOnly</td>
<td>180</td>
<td>180</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>split1, child1</td>
<td>FlexOnly with flex value = 2</td>
<td>220*2/3= 147</td>
<td>147</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split1, child2</td>
<td>FlexOnly with flex value = 1</td>
<td>220*1/3= 73</td>
<td>73</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split2, child1</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split2, child2</td>
<td>ContentOnly</td>
<td>80</td>
<td>80</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split2, child3-4</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split2, child5</td>
<td>ContentOnly</td>
<td>20+60=80</td>
<td>80</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split3, child1</td>
<td>FlexOnly with flex value =1</td>
<td>80</td>
<td>80</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split3, child2</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split3, child3</td>
<td>ContentOnly</td>
<td>80</td>
<td>80</td>
<td>False</td>
<td>20</td>
</tr>
<tr>
<td>split3, child4-5</td>
<td>ContentOnly</td>
<td>20</td>
<td>20</td>
<td>False</td>
<td>20</td>
</tr>
</tbody>
</table>
Algorithm 2 assignSizeTopdown(span)

1: Look up innerDefaultSize, outerActualSize for span
2: innerActualSize = max(innerDefaultSize, outerActualSize)
3: if innerActualSize > outerActualSize then
4: mark this span as scrolling needed
5: end if
6: for each split in splits inside of span do
7: totalChildrenODSizes = sum over outer default sizes of all children spans in split
8: totalChildrenWeights = sum over flex values of all children spans in split
9: paddingAmountNeeded = innerActualSize − totalChildrenODSizes
10: if paddingAmountNeeded > 0 then
11: divide paddingAmountNeeded among the flex children according to their flex weights
12: if no flex children are present, then all of paddingAmountNeeded should go to the last child
13: outerActualSizeOfChild = outer default size for child (from look up) + padding to child
14: save outerActualSizeOfChild
15: else
16: for each child in children do
17: outerActualSizeOfChild = outer default size for child (from look up)
18: save outerActualSizeOfChild
19: end for
20: end if
21: end for
content centered. The strategy we used to center any element was to add a span to its left (top) and right (bottom), and assign them to FlexOnly with the same flex value. For example, on the login page, the main content should stay exactly at the center vertically, so we create a column span to its left and right, and set a FlexOnly setting of flex:1. Thus, both columns expand and shrink down to zero as needed, and the rest of the spans remain as contentOnly.

Starting from the home page, we have a menu bar on top. The strategy we used to build the menu bar is to divide using a fixed split into different column spans (Figure 5-6). The first division is set to contentOnly and contains the name of our application, Espalier. The second through fifth divisions are all allocated for empty space in the menu bar. The second span is positioned to the left of the span, which contains the search label and we would like to for it expand or shrink to zero as needed so we set it to flexOnly. The search label column is the third span, while the fourth is the search bar column – both are set to contentOnly. The fifth is the other column span set to flexOnly which we would like to give a flex value equal to the second span so the third and fourth span can always be centered. This makes the menu bar shrink and expand as needed while also making sure the content below is centered. A row span is added before and after the search bar to add some padding to separate the search bar from the menu and list of products below. We set these two spans to exactOnly with exact: 50 px. We can leave the row span which contains the products to be contentOnly because it is the last child span of the parent fixed split, so all extra spaces by default would go toward it.
Figure 5-6: Product Page in Default View. The arrows indicate the children of the fixed split. The yellow arrows are children 2 through 5, which are set to FlexOnly so that they can expand and shrink as needed. The white arrows indicate the ContentOnly children spans.
Chapter 6

Evaluation

6.1 Replicating the Facebook Home Page

To demonstrate and evaluate the current styling functionalities of Espalier in the real world, we chose to replicate a well-known web page that has a variety of sizing and styling constraints – the Facebook home page, as it is a well-known page with a variety of components. Figure 6-1 below is a screen shot of the original Facebook homepage with the names and faces of actual users grayed out and components labeled. Our replica contains all of the above components and attempts to provide the same user experience and layout as the original homepage. Below are screenshots of how our application would appear on three different screen sizes: large (full screen on desktop), medium (full screen on laptop), and small (about 70 percent of full screen on laptop). Figure 6-2 is how our application would like on a large screen. Figure 6-3 is how our application would like on a medium screen. Figure 6-4 is how our application would like on a small screen.

As the figures demonstrate, we have largely reproduced the page using Espalier. In the following sections, we will provide more details as to how the application is built, as well as any compromises we made while attempting to replicate certain behaviors.
Figure 6-1: Original Facebook Homepage with grayed out friend’s names and photos. The labeled components are: 1. Menu bar 2. Main container 3. Left navigation menu 4. Main content area 5. Events/stories area 6. List of friends 7. Create Post section 8. List of friend’s posts. Note that the list of friends is only visible when window width is large enough to fit all contents in

6.1.1 Overall Structuring

We attempted to structure our replica by reproducing the different components as diagrammed in Figure 6-1 using fixed splits. We added a fixed row split on the root row span to divide the page into the blue menu bar and main container. The blue menu bar’s span is then divided vertically using a fixed split into column spans that hold the blank space, the Facebook icon, the search bar, another blank space, current user’s profile photo, and the menu options. We then defined another fixed column split on the root column span to split the main container whose scope picks up from where bottom of the menu bar and covers the rest of the page. The children column for the main container includes left navigation menu, the main content area, the stories/events area, empty space columns and the friends list. The next few subsections provide details on the challenges we encountered while trying to replicate the sizing behavior of these components.
(a) When page is loaded

(b) User scrolls down

Figure 6-2: Large Screen
(a) When page is loaded

(b) User scrolls down

Figure 6-3: Medium Screen
Figure 6-4: Small Screen
6.1.2 Priority Flex

Both the menu bar and the friends list require a type of behavior which we call "priority flex". For the menu bar, when the window width shrinks, the original Facebook app responds by shrinking the blank space column spans, until they have shrunken to their minimum size. In Figure 6-5a, we can see that the original Facebook home page shrunk the blue spaces on both ends as well as the middle section, but the sizes for other components are unaffected. As the width shrinks further, the original Facebook page starts shrinking the search bar until the search bar is reduced to its minimum size of about 100 px (Figure 6-6a).

The friends list has a similar behavior. As the window size shrinks initially, the gray columns in-between the stories area and the friends list, and to the left of the navigation menu, shrink to their minimum. All other content remains the same (Figure 6-7a). As the window width continues to shrink, there does not exist space to fit the entire friends list even after the empty columns are shrunken to their minimum. At this point, the friends list disappears entirely and the empty columns expand to fill up the leftover space that was previously occupied by the friends list (Figure 6-8a).
(a) Original Facebook home page with smaller window width. We can see that empty columns shrink to their minimum size, while the friends list is unchanged.

(b) Our replica’s home page with smaller window width. We can also see that empty columns shrink as well as the friend’s list.

Figure 6-7: Comparison of the Friends List on a Small Window Width
(a) Original Facebook home with even smaller window width. We can see that the friends list disappears and the empty columns expand back to take up the space left by the friends list.

(b) Our replica with even smaller window size. We can also see that empty columns as well as the friend’s list further shrink.

Figure 6-8: Comparison of the Friends List on a Further Decreased Window Width
In our replica, we were not able to reproduce this exact behavior. By definition, the flex spans divide the left over space based on their flex values at the same time. We do not prioritize which spans should shrink first among all other flexible spans, so all spans respond at the same time. Furthermore, we cannot specify a condition for when the size of a span should completely collapse to zero. The most similar results we were able to obtain was where all the corresponding spans were assigned appropriate flex values based on their observed rate of expansion. In our examples, both the search bar and the empty spaces shrink from Figure 6-5b to Figure 6-6b and both the friends list and the empty columns shrink from Figure 6-7b to Figure 6-8b. To add support for this in the future, we would need to change the assignSizeTopDown function to pad according to an ordering of the flex spans that users could specify.

6.1.3 On Hover Response

Currently, changing the style on hover is not supported. In the original application, when user hovers over each link, the background lightens and the borders appear around each navigation option. "On hover" is a powerful functionality because many styles changes might be desired for on hover, ranging from background colors to borders to visibility to changing text or icons inside. Thus, it would likely need to be built into our formula language.

6.1.4 Place Holders for the Main Content Area

Another feature we were not able to replicate was placeholder texts. In the original application, the text "What is on your mind, Willow?" is just a placeholder, and disappears when the user starts typing. In our replica, this is built using a formula that is the string concatenation between "What is on your mind, " and the current logged-in user’s name with a question mark. There is not a way to specify in our formula language to clear a cell on click. Thus, the text is unchangeable once set by a formula. An alternative way would be to add a procedure to a button which, when
pressed, clears that cell from existing formulas and lets users enter their thoughts. However, this seems very counter-intuitive so we chose not to do this in our replica. An appropriate solution may be to make placeholder text a label and simply leave a empty cell for users to write their thoughts into, similar to what was done for the search bar in the replica.

Currently, text of length zero and empty collections are explicitly presented. In part (b) of figures 6-2, 6-3, and 6-4 above, in Ann Chovey’s comment under the first post, we observe a striped area because there are no reactions. Additionally, in Figure 6-9, we note that the empty string where "is with" used to be as well as the striped spaces for friends’ names, reactions, and comments are empty. The empty string is there because in our structuring we chose to explicitly represent empty strings. We chose to represent empty string and empty collections explicitly for clarity of structure in the default and design view but this may not be appropriate for the unprivileged view.

6.1.5 Icons and Pop-up Menus

We currently do not have support for icons. The original Facebook uses icons for friend requests, messages, and other items. In our application, we chose to use links instead. Since we do support images, we could instead use an image in place of an icon, but we currently do not support binding an "on-click" event to an image. Alternatively, we could use a button, but there currently is not an existing strategy to customize the background of the button to have the look of the icon.

When clicking one of these icons in the original application, a menu pops on top of the current user interface, covering what is underneath. In the present structure of Espalier spreadsheets, we do not support overlapping sheets with different z-index. A single page would be built using a single sheet. Thus, due to the structure of our system, we cannot reproduce things like pop-up menus or chats. Since we chose to use links instead of icons here, when pressed, they will redirect the user to a different page in the application where it would show those menus.
6.1.6 Same Datatype Requirement for Collections

The friends list is a variable split and its parent span has a fixed column split defined on it to divide the friends list into profile images, names, and last active times. Note that we used only text to represent the last active time here. In the original application, if the user’s status is currently active, there would be a green dot next to the user’s name (Figure 6-7a). We were not able replicate this feature because, by definition, each friend is an instance of the friends collection and our structure requires that the type of a property does not vary between different items in the same collection (the last active time of a friend can be of only a single datatype).

We run into similar limitations again with the reactions to friends’ posts. In the master dataset sheet, each post has a collection of reactions. Due to the instantiation process, all items in the same collection get exactly the same style. So when displaying the numbers of reactions after the post’s body, we can not style each reaction name a different color.
6.2 Analysis

In summary, all options on the styling were used when replicating the Facebook home page. We were able to replicate the display and behavior of most critical components. At the same time, we were not able to build an exact replica and made some compromises as needed to achieve the same functionality.

These compromises had to be made mostly because certain desired behaviors were not fully reproducible using the styles we currently support. This situation can be solved, however, when we add more functionality to our system in the future. One such behavior would be to have more graphical elements. For example, we may need to add support for icons and styling the text and background of buttons. Another behavior is priority flex, in the example of the friends list. Currently, all children spans which are set to be flex are considered to be at the same level – that is, they all shrink and expand together. We will need to add a certain way to prioritize which spans should shrink first, before attempting to shrink other spans. A final desired behavior is to allow stylistic changes for on-click or on-hover events in the example of the left navigation menu and the placeholder text. We would need to modify the formula language to enable conditional and on event changes in styles.

Two compromises were made due to the structural restrictions we chose to place on Espalier. These restrictions were placed for better data consistency, organization, manipulation, but they do limit us in certain cases when combined with styling. These cannot be solved by adding more complexity in the styles we support, and will likely require a change in structure. One desired behavior was to have a menu pop-up on top of the current page, or have reaction icons partially cover the comment those reactions belong to. Our structure only supports a single layer of the spreadsheets, not multiple layers on top of each other. Another desired behavior was having different types of content for the same field of an object type. In the above example, we want a green dot when a friend is active now, but a time in text form if the friend
is not active. We are unlikely to change the structure of our spreadsheet in order to compensate for styles.
Chapter 7

Conclusion

We built Espalier as a new programming paradigm that rethinks the computational model of spreadsheets. The existing scoped-splits model offers it as a useful tool for presenting and manipulating structured data. With the already wide capabilities of its structuring model, this thesis has focused on augmenting it with styling capabilities, so as to allow users to manipulate data and style in place, which may allow Espalier to be used as an app builder.

Styling is composed of mostly two separate contributions: first, styling the design view after necessary spans are added. Second, building the styling menu and allowing users to specify both allocation and span styles throughout it.

To demonstrate the capabilities of styling, we added styling to our previously-built store example and conducted a case study of replicating the Facebook home page using our styling tools. We found that our styling strategy is sufficient for basic websites and desired behaviors. For complex sites, such as Facebook, we were not able to replicate the exact depiction or behavior, but were nonetheless able to substitute similar displays and behaviors, using the current styling parameters we support. In the previous chapter, we discussed cases of which our styling is insufficient, though are mostly solvable by adding more styling support, which we plan to build in the future. A few difficult cases were due to the features of underlying structure propa-
gating certain constraints to the styling, but we will likely not change our structure to compensate for styling.

7.1 Future Work

In the future, we would like to add the following capabilities into our styling:

- Priority Flex, which will allow users to specify an order of importance to which spans should expand or shrink first before others.

- Styling options in our formula language, which are currently only used for structural changes.

- Pop-up menus, which may be used for components such as drop-downs or sub-menus.

- Methods to specify changes in styling that are event-triggered. One example would be on hover, change background to a different color.

- Adjusting opacity for background colors or images.

- Button styling including adding background colors and images

- Icons.

With the augmented styling capabilities, we anticipate that, in the future, Espalier will be used as a preferred app-building tool capable of building and styling complex websites, with both ease and flexibility.
Bibliography

