

Reducing Authoring Complexity on the Web

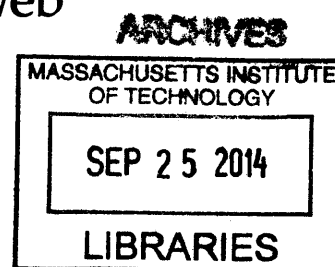
with a

Relational Layer for Web Content

by

Edward Oscar Benson

S.M. Computer Science and Engineering, Massachusetts Institute of Technology (2010)
B.S. Computer Science, University of Virginia (2005)



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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author
Department of Electrical Engineering and Computer Science
August 29, 2014

Signature redacted

Certified by
David R. Karger
Professor of Computer Science and Engineering
Thesis Supervisor

Signature redacted

Accepted by
Leslie A. Kolodziejski
Professor of Computer Science and Engineering
Chairman, Department Committee on Graduate Students

Committee Members
Tim Berners-Lee, Professor
Rob Miller, Professor

Reducing Authoring Complexity on the Web with a Relational Layer for Web Content

by

Edward Oscar Benson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
at the Massachusetts Institute of Technology

August 29, 2014

Abstract

When we browse the web, we experience rich designs and data interactivity. But our creation efforts of such content are often hampered by the great engineering effort required. As a result, novices are largely limited to authoring read-only content from within content management systems, and experts must rely on complex software toolchains like Ruby on Rails to manage software complexity. These application-level strategies have enabled tremendous creative output, but they only alleviate, rather than eliminate, core sources of web authoring complexity.

This work shows that adding a declarative, relational layer to the web stack reduces the complexity of authoring and reusing web content by providing a way to reason about how structures on the web fit together and what should happen when they change. For static content and design, I demonstrate new and more usable authoring and deployment techniques. For dynamic content, I demonstrate how a relational layer can transform HTML and spreadsheets into read-write-compute web applications without any procedural programming at all. User studies show that HTML novices can learn to apply these techniques in only a few minutes, increasing their creative capacity beyond read-only rich text. And professionals can use this approach to drive a development process based on full-fidelity design mockups rather than code fragments.

Thesis Supervisor: David R. Karger

Title: Professor

For Grace

Acknowledgements

When reading the acknowledgments section in the theses and books of others, I've always skimmed straight over what always seem to turn into a long list of names. But now I understand how much gratitude and happiness comes from getting to type each of your names and remember, for each of you, the times we've spent together and what I've learned from you. I am astounded at what a brilliant, caring, opinionated, daring, wonderful bunch of people I have had the privilege of spending these past six years with. Thank all of you. It has been such a wonderful ride.

To my wife Grace, I can not imagine these past six years in Cambridge without you. If I were to actually list all the ways you've been my source of strength and support, good humor and awesome adventures, it would require a whole chapter in this thesis. So instead I'll just say: I love you. Thanks for loving me.

To Mom and Russ, Dad and Tami, and Alex for your endless support and love.

To my advisor, David Karger. Few people I've encountered approach their work with the sheer degree of cheer, creative breadth, and good-natured contrarianism as you. I have learned a tremendous amount from you and have truly enjoyed working for and with you.

To my thesis committee members, Rob Miller and Tim Berners-Lee, for their willingness to hear my ideas and push me in directions I hadn't thought of.

To my dear friends from the lab Eugene Wu, Lydia Gu, Adam Marcus, James Cowling, Grace Woo, Szymon Jakubczak, Meelap Shah, Neha Narula, and Michal Depa. And those from down the street: Stephanie Maximous, Noah Rindos, Benn and Daniela Egan, Chad Benedict, Jerre Maynor, and Caitlin McCollister. You are all ridiculous people, and I love you for it.

To my fellow Haystackers, thanks for the friendship, advice, and camaraderie: Eirik Bakke, Anant Bhardwaj, Amy Zhang, Katrina Panovich, Max Van Kleek, and Michael Bernstein. Friends from the UID group, always willing to chat and lend advice: Elena Glassman, Carrie Cai, Philip Guo, Juho Kim, Tom Lieber, Kyle Murray, Max Goldman, and Jones Yu. My DIG group buddies, Fuming Shih (the first person I met at MIT), Oshani Seneviratne (thanks for sticking with me in the cathedral office!), and Ilaria Bacardi. To the NLP group, especially Harr Chen and Aria Haghighi, for teaching me all the math I know, and also Christy Sauper, Branavan, Tahira Naseem, Nate Kushman, Yoong Keok Lee, and Regina Barzilay for taking a chance on me. And to all you folks whose groups I didn't pretend to be in, but who

have been wonderful friends and colleagues nonetheless: Tim Danford, Jean Yang, Aditya Parameswaran, Leilani Battle, Eunsuk Kang, Jackie Lee, Andrew Sabisch, William Li, Neha Gupta, and Ramesh Sridharan.

To the whole W3C team, particularly Amy van der Hiel, and also Sandro Hawke, Eric Prud'hommeaux, Veronica Thom, Jérémie Astori, Andrei Samba, Ralph Swick, Judy Brewer, and Maria Auday. Thank you for so kindly welcoming me into your space to work and hang out with you these past years.

To fellow summertime conspirators and dreamers of big ideas, even if they didn't always pan out exactly the way we'd imagined: Chaki Ng, Andrei Villarroel, Dan Lorenc, Kwan Hong Lee, Ed Hallen, and Ryan Choi.

To the UROPs I've had, you guys are awesome: Jessica Andersen, John Wang, Oliver Song, Tochukwu Okoro, Connie Huang, Sarah Scodel, Jason Yonglin Wu, Daniel Ronde, Zach Wener-Fligner, and Aizana Turmukhametova.

To all the mentors from CSAIL and elsewhere who have gone out of their way to help me. Sam Madden, who, a few days before my first paper deadline, showed up in my office, put his feet up on a spare desk, and said, "I've cleared my schedule. How can I help?" To mc schraefel (lowercase intentional) and Jay Borenstein, for their guidance in research and career. Jenny 8. Lee for always bringing a tornado of ideas, advice, and encouragement into town. Martin Wattenberg, Fernanda Viegas, Jon Orwant, Michael Riley, and really the whole Google Research team, who have been so tremendously kind and bright.

And thanks, finally, to everyone at CSAIL and EECS who work so tirelessly to make this place run. To the whole TIG team, who are really more like an elite commando unit than an IT department. And thank you Janet Fischer, Alicia Duarte, Colleen Russell, Be Blackburn, Patrice Macaluso, Marcia Davidson, and Rebecca Yadegar for all your help over the years.

What a lucky guy I am to have all of you in my life.

Contents

1. Introduction	13
1.1. Some Reasonable Goals	14
1.2. Meeting Those Goals (A Preview)	16
1.3. Web Authoring Practice Today	18
1.4. The Case for Revisiting the Declarative Stack	21
1.5. Thesis Approach	22
1.5.1. Thesis Statement	22
1.5.2. Assumptions and Limitations	23
1.6. Contributions	25
2. Related Work	27
2.1. Structure and Web Documents	27
2.1.1. Hypertext Markup Language	27
2.1.2. Cascading Style Sheets	29
2.1.3. Extending HTML	31
2.1.4. Adding Semantics	32
2.2. Authoring and Reusing Web Documents	33
2.2.1. Platforms for Reuse	34
2.3. Authoring Data Visualizations	38
2.4. End-User Front-end Language Design	40
3. Case Study: The Exhibit Framework	41
3.1. Methods and Dataset	43
3.2. Framework Design: the "Star Model"	44
3.3. Exhibit Authors	47
3.3.1. Author background and needs	47
3.3.2. Authoring Approach	49

Contents

3.4. Authoring Data	52
3.4.1. No love for JSON	52
3.4.2. Diverse but Human-Scale Datasets	53
3.4.3. (Ab)using Spreadsheets to author non-Tabular Data	55
3.5. Visualizations	57
3.5.1. Navigation, Visualization, and Just Plain Publishing	58
3.6. Publishing	59
3.6.1. Platform Agnostic Client/Server Extensibility Needed	59
3.6.2. Publishing Environments a High Cost	60
3.6.3. Exhibits need Wide Layouts	61
3.6.4. Exhibits tend to be Stand-Alone	61
4. Cascading Tree Sheets	65
4.1. An Intuitive Introduction	66
4.2. Model	69
4.2.1. Trees	69
4.2.2. Relations	73
4.3. Surface Form	73
4.3.1. CTS Statements	74
4.3.2. Tree Sheets, Style Blocks, Inline, and Imports	75
4.3.3. Statements versus Relations	76
4.3.4. Default Trees and Late-Binding Trees	77
4.4. Semantics	77
4.4.1. Choosing what concepts to represent	78
4.4.2. Fitting Template Operations onto a Relational Model	80
4.4.3. Graft --- Structured Content Equivalence	82
4.4.4. Are	83
4.4.5. If-Exist and If-NExist --- Existential Dependency	86
4.5. Example Scenarios	87
4.5.1. A Basic is Relation: ``This this is like that thing.''	87
4.5.2. A graft with a nested is: ``Including a template to wrap my data.''	88
4.5.3. A list of Todo items	90
4.5.4. Hiding Completed Items	91
4.5.5. Composing Multiple Trees	92

4.6. Rendering Web Pages	93
5. Static Content Authoring	95
5.1. Authoring Efficiency and Mockup Driven Development	97
5.1.1. Thinking about HTML Authoring as Signal-to-Noise	98
5.1.2. Mockup-Driven Development	100
5.1.3. Content Authoring User Study	103
5.1.4. Summary	109
5.2. Reusing Content and Presentation	109
5.2.1. CTS Microformats	110
5.2.2. Widgets	112
5.2.3. CSS Shims as Simple Widgets	112
5.2.4. Themes and Retargeting	118
5.3. Maintaining Code: Proper Encapsulation	124
6. Reactive Programming, Data, and Computation	129
6.1. Spreadsheet Backed Web Applications	131
6.2. An Example: A Spreadsheet-backed Microformat Widgets	132
6.3. Incorporating Spreadsheets into CTS for End-User Programming	133
6.3.1. Casting Spreadsheets as Trees	133
6.3.2. Simplifying the Language	134
6.3.3. Spreadsheet Selector Language	135
6.4. Synchronizing Data as it Changes	137
6.4.1. Generating Transforms	138
6.4.2. Applying Transforms	139
6.4.3. An Example Sequence of Transformations	140
6.5. User Study	141
6.5.1. Result: Learning and Applying Quilt	144
6.5.2. Designing Spreadsheets to Back Web Apps	146
6.5.3. Debugging Behavior	147
6.5.4. Overall Reactions	148
6.6. Discussion	150
6.6.1. Low level end-user programming	150
6.6.2. Design Patterns	151
6.6.3. Copy-Paste-Modify	153

Contents

6.6.4. Visual Programming	153
6.6.5. Future Work	155
7. Discussion	157
7.1. Can this Relational Model Build Non-Trivial Applications?	157
7.2. Are Other Relation Sets Viable?	161
7.3. Does the Schema Alignment Problem make Post Hoc Retargeting just an Academic Exercise?	161
8. Conclusion	163
Appendices	165
A. CTS Grammar	165
B. Formal Rendering Engine Description	168
B.1. Preliminaries	169
B.2. A Notation for Unranked Relation-Aware Transducers	169
B.3. The Rendering Process	172
B.4. Step 1: Alignment	172
B.5. Step 2: Filtering	173
B.6. Step 3: Combination	175
B.7. Summary	176
C. Rendering Examples, Under the Hood	176
C.1. A Basic is Relation: "This this is like that thing."	176
C.2. A graft with a nested is	177
C.3. A list of todo items	179
D. Lossless Rendering Proof	180

1. Introduction

Web sites created by professionals are filled with interactive functionality, are styled with sophisticated layout and design, and are tailored to diverse domain-specific information. These attributes are not just shallow beauty: they help people manage their own information and communicate it effectively to others. But these web sites are increasingly costly and difficult to build.

When professionals create a web site, they find that they spend great time and effort writing and rewriting code that might have otherwise been reusable. This adds up to considerable effort. The United States Healthcare.gov website, an insurance information portal, infamously \$18 million dollars to create. But even smaller jobs requiring only web design, such as a custom theme for a the Drupal content management system, commonly cost many thousands of dollars. Once these sites are created, experts can not easily reuse the code for other purposes. A custom Drupal theme, for instance, can not be used with WordPress, an alternative content management system.

When novices creates web content, they find that they are unable to produce the same kind of experiences and capabilities that they are used to consuming. Instead, they are limited to Content Management Systems (CMSs) which boil down web publishing into typing rich text into a box and pressing a *Publish* button. But using CMSs to mediate web authoring comes with many compromises. CMSs require additional labor or money to install and maintain application servers and database systems—essentially administrative overhead from the perspective of content creation. And CMSs rigidly limit creativity in order to optimize the publishing pipeline. They have poor support for data models other than blog posts and text pages, and require programming skills to create new designs (often encoded as PHP-based theme files) to style content. Finally, CMS data storage, plugins, and designs are usually incompatible with other systems, locking users into a particular vendor.

The fundamental problem this thesis addresses is that creating, sharing, reusing, and maintaining rich, read-write web content is often out of reach for novices and costly for

1. Introduction

pros.

Addressing this problem has benefits far beyond the confines of a browser window. The web is a place for culture and storytelling, and also a platform for serious software engineering. Newspapers and magazines are increasingly web-first and print-second, all major eBook formats are derivatives of HTML, and social media largely takes place using web formats. On the engineering side, web-based office software is displacing its desktop counterpart, and web formats are increasingly viable as an intermediate representation layer for platform-independent mobile applications.

Storing, manipulating, and presenting information are some of the oldest practices of society. The ease and productivity of these practices is both a measure of and contributor to our progress. While the gold rush to get on World Wide Web may be over, the reach of the web platform only continues to grow in importance, often under the guise of different names. Underneath, the two fundamental formats which describe web content---HTML and CSS---act as a critical part of society's infrastructure. This thesis improves upon the underlying infrastructure of the web to support authoring by both novices and professionals.

1.1. Some Reasonable Goals



Figure 1.1.: (a) The HTML powering a popular webmail client, under the hood. (b) A gallery of web designs. (c) A faceted search over structured data.

To help motivate the approach this thesis will develop, let's consider three basic activities on the web that authors both want and need to do. Figure 1.1 shows images representing these activities.

1.1. Some Reasonable Goals

1. *Write and edit web content without hassle.*

It is often useful to edit a site by hand or explore how someone else created theirs, but examining the source of professionally created sites often reveals HTML that is beyond even the ability of an expert to understand without the help of tools, like the code for an email in box show in Figure 1.1 (a). This work develops methods for separating web pages into separate minimal content documents and design mockups, improving the usability of working on both.

2. *Easily reuse web designs found elsewhere.*

Instead of editing HTML by hand, users often turn to CMSs, which provide a GUI and relieve the need to edit HTML source. But CMSs only permit web designs written as software plugins, chopped up into multiple files, rather than ordinary HTML. This makes it difficult to reuse pages found on the web, like the design gallery in Figure 1.1 (b). This work will show how to reuse designs—with or without a CMS—by authoring a few CSS-like statements that connect a page on the web to the page you want to theme.

3. *Weave structured data into the pages they make.*

Authors have all sorts of data that does not fit the model supported by content management systems and their themes. But creating web applications that manage custom data objects currently requires custom programming and database administration. Authors want to theme their objects, search through and filter them (like the facets in Figure 1.1), add new objects, and edit old ones. This work will show how accomplish these goals with just ordinary HTML and a Spreadsheet.

Not all data fits the “posts and pages” schema of a CMS. But managing a site that permits linking to and then searching over it, like the facets in Figure 1.1 (c), or supporting currently requires custom code and database administration to handle the non-standard data shape.

These three goals illustrate three common needs of both novices, who may author via a CMS, and experts, who may be more familiar with source editing.

1. Introduction

1.2. Meeting Those Goals (A Preview)

This thesis develops methods for accomplishing these goals at very base of the web stack so that the solution can be enjoyed by source authors, programming frameworks, and user-facing applications alike. The core idea is the introduction of a declarative, symmetric, relational layer to supplement HTML and CSS. This relational layer enables authors to make statements about how structures on the web (a DOM node here, a JSON array there, a Google Spreadsheet cell over yonder) relate to each other. In Chapter 4, we will develop a specific set of relations that act as the relational equivalent of a templating language. The Appendices of this thesis provide the details of a calculus for processing over these relations to produce web pages with some very interesting information-preserving properties, that we dub *lossless rendering*.

We stick all those implementation details in the Appendix so that we can move straight into the ramifications of adding this new layer to the web. From a functionality perspective, this work shows that this relational approach provides sufficient information to accomplish many of the tasks we already perform, including web templating, reactive programming, widget packaging and invocation, and site theming. But this work also shows that this approach improves the usability of these tasks compared to current practice. It offers better encapsulation, new mockup-driven development paradigms, and easy read-write-compute web authoring all without client-server programming.



Figure 1.2.: The *Twenty Eleven* WordPress theme (A) ported to two different CMS systems. This porting process requires rewriting the multi-file PHP code package WordPress uses to encode a theme in the respective language of the other systems. This thesis makes such work unnecessary.

As a small taste, consider the design reuse goal you just read in the previous section. With current methods, adapting any of those hypothetical designs for use with a content management system requires coding, a custom plugin of sorts that

1.2. Meeting Those Goals (A Preview)

we happen to call a theme. Figure 1.2 shows an example of this—the same theme, called *Twenty Eleven*, offered separately for the WordPress, Drupal, and Blogger. Each instance of this theme was hand-written in its respective platform's own custom theme programming language.

The techniques and tools developed in this thesis will enable a different approach, in which the theme is simply published as an ordinary HTML design mockup. That design mockup can then be symmetrically mapped onto different pages and data structures on the web. This enables the design mockup to act as a theme for display, and it also enables edits to the resulting document to automatically trickle back to data updates in the source documents.

Here is an example of what this mapping looks like in a language called **Cascading Tree Sheets (CTS)**, which this thesis develops. The code below maps a hypothetical HTML design onto a JSON data structure:

```
@html design http://www.example.org/FancyBlogDesign.html
@json my      http://www.mypage.com/blogFeed.json

my | title           :is      design | h1#title ;
my | subtitle        :is      design | h2#subtitle ;

my | articles        :are     design | section.posts ;
my | articles.*.title :is      design | h2.post-title;
my | articles.*.body :is      design | div.post-body;
my | articles.*.author :if-exist design | div.post-credits;
my | articles.*.author :is      design | div.post-credits > .name;
```

This example has JSON selectors on the left, CSS selectors on the right, and relation types in the middle. The two sides could be any hierarchical data type, though, as long as it responds to a selector language.

A useful big-picture analogy can be drawn to the advent of relational databases. The act of overlaying binary blobs on disk with a relational schema and the rules of relational algebra enabled a wide variety of operations: querying, merging, updating, optimizing. Similarly, this thesis shows how the presence of relations like those above, and a calculus for operating over them, results in benefits that ripple all the way up the web programming stack.

1. Introduction

With this glimpse of the goal-post in mind, let's return to the discussion of current web programming practice in order to better situate this work.

1.3. Web Authoring Practice Today

The needs of web authoring today are met with a complex authoring and development toolchain that automates much of the low-level work required to construct and publish a web site. This toolchain includes components used only to author (such as intermediate computer languages that compile to web languages), components used to publish a site (such as server-side frameworks), and components that provide end-to-end solutions (such as content management systems).

It is useful to make a distinction between two types of authoring: static and dynamic. In this thesis we use slightly unorthodox definitions of *static* and *dynamic*. Normally static content refers to raw HTML files, served as-is, and dynamic content refers to programming machinery that spits out HTML on demand (e.g., a blog template). Here we will use the terms from the perspective of the web reader. Static will mean read-only (like a news article), and dynamic will mean read-write-compute (like a social media site, or a search-able library catalog).

Static Authoring Static Authoring is used in this thesis to mean the authoring of content and design that does not include user inputs at runtime. That definition means this thesis considers dynamically rendered templates "static content."

At heart, static web authoring is HTML and CSS authoring, but developers today often use intermediate representations that are only transformed into HTML and CSS at the last minute. CSS preprocessors like LESS [79] and SASS [19] assist developers in wrangling the thousands of lines of CSS required for a modern layout, adding concepts like variables and conditional statements to CSS's otherwise declarative format. And templating frameworks like EJS [10], Jade [1], Velocity [1], and Mustache [92] provide similar preprocessing for HTML, expressing it in an alternative format that eventually, at run time, is turned into HTML.

These HTML templating frameworks are used even for sites that do not involve dynamic data queries and databases. The "static blogging" community, for example, is a thriving community of web authors who use sophisticated content preprocessing pipelines to compile an entire web site that, at publication time, consists

1.3. Web Authoring Practice Today

entirely of HTML and CSS files. Template engines and CSS preprocessors are used to define things like page templates and style sheets once, and then re-apply them to each particular content page. It thus is entirely possible that a hand-built static web application today does not include a single file in its project directory that is independently renderable as a coherent web document.

This more of authoring may be one way to it speedier to generate HTML. But it also results in several problems down the road. The chief problem is that they provide macros to automate HTML authoring and manipulation without providing end-to-end encapsulation and complexity management. For example:

- While frameworks like LESS and Velocity treat CSS and HTML as an assembly language to be compiled into, they do not save the author from having to understand and manipulate assembly-level information. Even though template files may be split apart on the server, in the browser the author must write CSS and Javascript that addresses the single, merged version.
- Designs authored in modern template languages are renderable in a web browser or editable as raw HTML. This is because they commingle HTML with other programming language and often encourage use of fracturing templates into small fragments, stored in separate files, for dynamic inclusion. This means that the user interface must go through a manual conversion process from design artifact to production artifact, placing a barrier between designers and the systems they design.
- Template languages are optimized for machine-centric data formats like JSON, Ruby objects, or Python objects. Because of this even if the templates could be rendered in a browser, they could not be tested with real data unless that data was prepared in one of these formats, which may also require a running application server. The latter adds complexity to the design process and the former (JSON) has poor usability for human editing (for example, no multi-line strings) and is disliked by end-user web authors [7].

In contrast, the relational techniques developed in this thesis will provide an alternative method of handling static content and design that also solves these problems. CTS will manage complexity not just by enabling reuse of content, but with end-to-end encapsulation of that content. And both the designs, mock content, and live content when using CTS can be ordinary, unmodified HTML.

1. Introduction

Dynamic Authoring Dynamic Authoring is used in this thesis to refer to the authoring of web pages that (1) allow the web user to create, modify, and delete data, or parameterize views, and (2) perform and display computation on that data. Examples of this type of dynamic authoring include the comment section on a blog, a survey where users enter data, or tallying up votes to show in a bar graph.

Like static authoring, dynamic authoring today is accomplished using extensive toolchains. In the client, frameworks like Meteor [66] and Angular [37] provide Javascript programming environments that assist programmers in manipulating network connections, controller logic, and HTML manipulation. These client-side frameworks stay in constant communication with their counterparts on the server, frameworks like Node [76], Ruby on Rails [41], and Django [50].

These extensive toolchains help automate the tasks of querying databases, invoking template engines, accepting incoming connections, listening for browser events, and making Ajax requests. But they do so in a piecemeal fashion that leaves applications with lots of code and little reusability. For example:

- Every new form upload from the client requires a custom handler on the server to listen on a particular address, parse form data, and make database calls to insert a new row.
- The code which produces a data-driven page does not pass through enough information to the browser to do anything with the rendered data. Something as simple as allowing a web reader to edit the title of a blog post requires custom Javascript event handling, Ajax calls, and custom hooks on the web server.

No wonder services like Google Forms are necessary. Simply accepting data from a web page and appending it to a list, with today's architectural styles, requires tremendous technical knowhow.

The techniques in this thesis will show how many of these issues which cause so much coding essentially amount to synchronizing data structures over a network connection. This synchronization process can be driven entirely by the relational annotations CTS provides. We show that this approach expands past just HTML synchronization and can incorporate other interesting data types, such as spreadsheets.

1.4. The Case for Revisiting the Declarative Stack

What all of the authoring scenarios in Figure 1.1, and the common practices in Section 1.3 have in common is *managing structure*. Consider the example of the recipe website: the recipes themselves are structured information objects, as are the templates that organize them on the page and the CSS that styles them. The author of the web application must decide how to store all these separate structural elements: in one document? in many? in a database? The problem quickly grows complex: data format and programming quickly get involved as well, and each choice constrains the way assets can be reused and editing.

Given the complexity of this process, it is no wonder that content management systems dominates online publishing. CMSs automate the common case—pages of text, organized in lists—so that authors can focus on writing content. While these tools have done much for the state of publishing, they also represent short-term fixes to larger challenges facing web publishing.

The web is a much larger and more important platform than any one application, such as Wordpress, Drupal, or MediaWiki. If we target the application layer as the location from which web authoring enhancements originate, the result will not be a better web, just scattered pockets of application features. By pursuing ways to manage the structure of web documents *at the source level*, solutions we find can universally benefit the web platform regardless of which application-layer tools one wishes to use.

The creation of Cascading Style Sheets (CSS) provides a good example [63]. Soon after the introduction of the web, authors began coercing HTML's `<table>` and `` elements to create web pages with designs far more complex than mere "hypertext documents." Achieving these complex designs with HTML's paper document-centric tag set resulted in HTML so cluttered with stylistic tricks that it became unmanageable. Cascading Style Sheets solved this problem by enabling people to annotate the DOM with style instructions provided in a domain specific language, with semantic addressing, and in an external file. Imagine how complex today's HTML would be if, instead of creating CSS, the community simply hid that style complexity behind applications that mediated authoring.

Cascading Style Sheets were a success, but now two decades later, we are faced once again with overwhelming structure management concerns. The list is extensive: reusing style, stitching together multiple web documents, offering WYSIWYG

1. Introduction

editing, transforming text or data into graphics, searching and filtering information, and so on. By addressing these concerns at the source level, via annotations to HTML documents, we can solve them for everyone in a domain-independent manner.

With a better annotative layer woven into the web, a home cook wouldn't need to pay for a developer to create a Recipe CMS just to publish recipes online. The mere act of annotating the existence and structure of a recipe inside an HTML document should be enough to infer all the theming, editing, and content management functionality required to manage a site. Because the cook's site is driven by text-based annotations, like CSS, other cooks who comes along could then easily copy, adapt, and republish that content and design without losing any of the structure.

Effective communication is not just about publishing text, but includes design, interactivity, and the curation and management of content collections. This thesis shows that addressing these problems at the source level, novices can more easily create and manage their own content and professionals can do so more efficiently.

1.5. Thesis Approach

1.5.1. Thesis Statement

My thesis statement is:

A relational layer atop HTML can enable native web support for many operations that are today handled at the application layer, improving the usability and capability of a broad range of authoring and management tasks for both static and dynamic content.

I organize the investigation of this hypothesis into four steps:

1. **Case Study.** A case study of Exhibit, a interactive data navigation framework driven by HTML annotations. I evaluate the experience of Exhibit users who have chosen this authoring approach over application-layer alternatives.
2. **A CSS for Structure.** I design and present a new web language called Cascading Tree Sheets (CTS) that adds the ability to talk about how structures on the web relate to other structures on the web.

3. **Static Content Authoring.** I explore, with analysis and user studies, how the presence of CTS enables usability improvement to static content and presentation authoring, reuse, and maintenance.
4. **Dynamic Content Authoring.** I explore, with analysis and user studies, how the presence of CTS enables create-read-update-delete content management functionality to be automatically layered onto a CTS-laden page. I design a system which enables end-users to create spreadsheet-backed web applications.

1.5.2. Assumptions and Limitations

To scope my research, I have made two key assumptions:

- *Improvements to lightweight content management on the web can be investigated independently of business logic that may be added later.* That is, improvements to the core structures which describe web content are useful by themselves---for those who seek only to publish and edit content, not to craft complex web applications. Further, improvements to this core content structure of the web are extensible to the many application architectures that power more complicated sites, such as a web store or a social network. As such, this work focuses on the core content management operations (*create, read, update, delete*) with limited attention to business logic extensions safe for a few key areas where the technology proposed has particularly interesting ramifications (such as Javascript widget design).
- *Addressing web authoring challenges at the source code layer ---rather than building more features at the application layer---is a worthwhile approach.* Two related push-backs I have heard against focusing so much effort on text-mode web programming are, "But we have WYSIWYG HTML editors and CMSs to do that stuff for us now!" and, "We should focus on improving tools so that people don't have to worry about raw HTML in the first place." This work makes the assumption that better document formats not only improve the experience of the many authors who choose (or are forced) to edit raw HTML, but they also have significant benefits for tool builders. Tools can leverage better incorporation of structure at the source level to provide even more powerful

1. Introduction

GUI-based features, and they can do it with the benefit of being compatible with other tools. As a quick example, WordPress themes are incompatible with Blogger themes, but were the aspect of "theming" to be solved at the HTML level, theme compatibility could be a fundamental assumption shared between CMS tools, not an application-layer feature.

1.6. Contributions

The contributions are:

1. **Case Study.** I curate a full life cycle dataset of visualization creation with the Exhibit framework and show that this declarative model of importing and relating HTML content meets a wide array of information management needs. I additionally uncover new information specific to the Exhibit tool, such as data authoring and visualization design habits.
2. **A CSS for Structure.** I develop the notion of a "relational web language" which enables authors to express how different structures on the web relate to each other. I develop a language called Cascading Tree Sheets (CTS) to experiment with this language and a Javascript runtime interpreter to use this language.
3. **Static Content Authoring.** I show how CTS enables a complete separation of concerns between content and design which improves the source-level content authoring process. I develop and evaluate a new authoring styles based on this principle, called Mockup Driven Authoring. This enables design artifacts to be reused as production artifacts and also allows *post facto* reuse of existing web sites as reusable design themes.
4. **Dynamic Content Authoring.** I show that CTS relations can be used to create read-write-compute applications by enlisting Spreadsheet software to serve as a data storage and computation layer. I perform a user study that demonstrates this method for authoring such applications can be learned by HTML novices. I describe design patterns and development questions piqued by using spreadsheets as an authoring medium for dynamic web applications.

A broader contribution of this work is the recommendation of a new standard for the web. We have many web standards related to data and document structure, but these two components only represent the web page once it is in the browser. By contrast, a tremendous amount of developer effort is expended forming the page for delivery to the reader and then getting information back out of the page to send to a server. Today we use a variety of application-level tools and conventions for these

1. Introduction

processes. **Standardizing a language that enables computers to reason about, automate, and optimize these processes would be a great benefit.**

Cascading Tree Sheets may not be the particular implementation that this standard forms around. But the general relational model this thesis develops, along with the reasoning behind its semantics and guarantees, is both concise in description and broad in application. As such, it appears to be worth serious consideration as a general framework on top of which a standard might be based.

2. Related Work

This thesis builds upon work from several communities, academic and industrial. This chapter provides an overview of this prior work. For the sake of consolidating commentary on key differences between prior work and this thesis work, forward references to concepts introduced in later chapters are occasionally made.

2.1. Structure and Web Documents

The web community is rooted in a deep history of work on document formats which attempt to balance between the needs of platform independence, authoring usability, and descriptive capability.

2.1.1. Hypertext Markup Language

HTML itself was a merger between the ideas embedded in the Standard Generalized Markup Language (SGML) [36] and hypertext [24]. SGML was a system for defining declarative, text-based document markup languages with a structure and vocabulary that would be familiar to today's HTML authors (bracket-enclosed tags such as <TITLE> and <H1>, for example). The key benefits of such an approach being that offered the ability to describe a syntax and vocabulary for document semantics—titles, sections, headings, paragraphs, and so forth—such that the encoding of the document was independent of the software packages that displayed or manipulated it. Hypertext was the notion of embedding links into documents so that readers could jump from one place (or document) to another rather than experiencing the document linearly.

The combination of SGML-style markup with hypertext, along with the systems software (the HTTP protocol [12] and the original WorldWideWeb application) to support a distributed ecosystem of publishing and interlinking, was enough to put

2. Related Work

HTML on the path toward being the *de facto* document format for modern publishing.

These precursors enabled HTML to quickly grow past its role as a way to create hypertext versions of paper documents. An entirely new medium of publishing, the "web page," was created. This new medium was not merely digitally-retrieved text documents but rather fully-native electronic documents designed with the web browsing software in mind: images to serve as navigational elements, forms to accept user information, content split across many linked pages, and so on. Rather than adopt the hierarchical "title, chapter, section, paragraph" style of text documents, web pages began exhibiting new styles of their own, influenced by the constraints and benefits that computer monitors and mouse-based navigation impose: navigation bars, side bars, page headers, and footers.

This change in style presented a technical challenge for web authors: how could the document-centric language of HTML be used to create these new interfaces? Anyone who developed web pages in the early 1990s remembers the answer well: `<table> <table> <table>`! Developers began using traditional HTML elements in creative ways, much like doctors sometimes prescribe medications for reasons other than their published purpose. The `<table>` element, in particular was extensively nested to organize content on the page to construct the standard web layouts that persist to this day. Colors, borders, dimensions, and fonts were added to these design-motivated structures using HTML attributes.

The trick worked, and a heavy period of design experimentation began on the web, but not without consequences. HTML described the kinds of text documents printed to paper with concision. But applying it as a language for graphic design resulted in low signal-to-noise ratio. Rather than organize text content, HTML `<table>` tags were used to scaffold designs. But so much design scaffolding was required that the "presentation HTML" obfuscated the "content HTML," making authoring and maintenance difficult. And because design instructions (e.g., font face, font size) had to be specified as element attributes, creating and modifying consistent designs across pages was arduous, with much repetition (e.g., specifying the font on each `<H1>`) and no global design context.

2.1.2. Cascading Style Sheets

Cascading Style Sheets (CSS) was created largely to manage this explosion of complexity [63]. CSS provided three key features:

- A logical separation of concerns: a language for describing the visual attributes of an HTML element separate from HTML itself
- A physical separation of concerns: the ability to consolidate these definitions into a file of their own and link to them from HTML documents.
- A semantic addressing scheme, so that these designs could be associated with elements based on class membership instead of just document structure.

The result of this addition to the web stack---when used properly---was dramatically improved editability of HTML documents, making "it easier to maintain sites, share style sheets across pages, and tailor pages to different environments" [96]. A design curation project that gained fame soon after the roll-out of CSS was the CSS Zen Garden [81], which demonstrated how a simple, fixed HTML document could be styled in a large variety of ways simply by linking to different CSS files (which designers would contribute).

The following years saw the balance between HTML and CSS stabilize. Additions and were made to both HTML and CSS, but these were largely incremental vocabulary additions rather than fundamental changes to the authoring workflow: additions like transparency, animation, 3D transforms for CSS and finer-grained document semantics like `<header>` and `<article>` for HTML.

But HTML and CSS together did not complete the separation of design from content concerns: a significant part of a web page's presentation cannot be described by CSS alone. It is instead defined by "presentational" HTML design scaffolding, as well as Javascript, interleaved with "content" HTML. Content HTML is wrapped in layers of presentational HTML to provide anchor points for CSS, or to control block-level layout beyond CSS's capabilities. As an example, consider the mass of HTML nodes that was needed just to place content inside a rounded rectangle before CSS3 introduced a single instruction for doing so.

Figure 2.1 shows one popular implementation of rounded corners with CSS2 [69]. This implementation requires 10 HTML nodes, 5 CSS classes, and 17 CSS properties

2. Related Work

to achieve, all for a single "atomic" piece of content. Today, rounded corners can be achieved with the single border-radius CSS property.

HTML	CSS
<pre><b class="b1f"> <b class="b2f"> <b class="b3f"> <b class="b4f"> <div class="contentf"> <div>Round FILL!!</div> </div> <b class="b4f"> <b class="b3f"> <b class="b2f"> <b class="b1f"></pre>	<pre>.b1f, .b2f, .b3f, .b4f { font-size: 1px; overflow: hidden; display: block; } .b1f { height: 1px; background: #ddd; margin: 0 5px; } .b2f { height: 1px; background: #ddd; margin: 0 3px; } .b3f { height: 1px; background: #ddd; margin: 0 2px; } .b4f { height: 2px; background: #ddd; margin: 0 1px; } .contentf { background: #ddd; } .contentf div { margin-left: 5px; }</pre>

Figure 2.1.: Before CSS3, painting rounded corners on an HTML element required significant effort, since replaced by a single CSS property.

We choose this resolved example to show the contrast between the before and after states. But no matter how many new CSS styling properties are added, there will always be new visual designs that require "presentation-only" HTML to accompany content HTML. Just sticking to corner style: what about picture frame-styled corners, drop-shadowed corners, or corners that zoom the container as the mouse comes near, and so on.

The root of the problem is that CSS only has the ability to make statements about individual elements. But many designs involve the coordination of many elements. Some require many elements because they have many fields of content, and HTML nodes are required to scaffold these different fields in a very particular way to support the presentation provided by CSS. Other designs have only one field of content but still need many HTML nodes because CSS properties alone are not sufficient to achieve the desired design around this content. Instead, many HTML elements in coordination, each individually styled by CSS, provide this design.

This situation with even individual content fields is inevitable, shown by a simple thought experiment. Imagine CSS rules exist to display n dots to the left of an HTML element. A designer can always create a design requiring $n + 1$ dots to

the left, necessitating an extra extra element containing a single dot to achieve this design.

CSS and HTML are thus incapable of fully separating content from design by themselves. This makes HTML documents harder to create, maintain, reuse, and tailor. When an author inspects HTML source, she finds large blobs of presentational HTML wrapping (and often obfuscating) meaningful content. To reuse markup, she must copy the entire blob, then find and replace the right pieces of content with her own. This might work for replicating an exemplar layout once, but what happens if an author wants to use the same layout repeatedly on many instances—for example, to nicely format each publication in a large list? The labor becomes substantial.

So, in short, HTML once suffered from runaway complexity and CSS swept in to solve it. Now, we are witnessing the complexity of web apps rise again due to the sophistication of our document structure and data interaction, but CSS alone can not solve these problems.

The Cascading Tree Sheets language presented in Chapter 4 provides a way to isolate and separate this structural scaffolding. Techniques for doing so are presented in Chapter 5, *Static Authoring*. The methods we show are modeled after the intuitions behind CSS but are targeted at wrangling structure, instead of style. CTS thus plugs the gap that CSS can not address, alleviating this new type of complexity we see today.

2.1.3. Extending HTML

Two primary extensions have been added to HTML that enable users to add their own attributes and nodes. These developments are important to note because the ability to extend a language is often the back-door through which a host of encapsulation and reuse capabilities can be introduced in without needing standards support. Cascading Tree Sheets, for example, uses HTML5 data attribute as a shim through which additional statements about document structure can be made.

XHTML is an XML-based variant of HTML with stricter rules of document construction but XML's extensibility as an added benefit [91] [64]. By defining a custom Document Type Definition (an SGML-borrowed way to describe the valid production rules of the language), authors can extend the valid tags and attributes of a web document. The Exhibit framework [48] uses this method, for example, to add

2. Related Work

attributes such as `ex:view` to specify that a DOM element represents the container of a data visualization, or `ex:lens` to specify that a DOM element represents the definition of a template for some abstract item type.

HTML5 introduced an alternative mechanism for associating data with HTML nodes, called "data attributes" [47]. Data attributes permit users to decorate DOM elements with arbitrary attributes as long as they are prefixed by the string `data-`. Data attributes open up a more limited avenue of extension than XHTML's DTDs because they only permit custom attributes and do not provide any way to specify namespaces or typecheck adherence to a schema.

2.1.4. Adding Semantics

<p>(a) Raw Data A first and last name expressed in RDF, using the Turtle syntax.</p>	<pre>@prefix foaf: <http://xmlns.com/foaf/0.1/> . <http://www.edwardbenson.com> foaf:nick "Ted" ; foaf:last "Benson" .</pre>
<p>(b) .. as RDFa in XHTML Identical RDF statements interwoven with an XHTML document using the RDFa extension.</p>	<pre><div xmlns="foaf:http://xmlns.com/foaf/0.1/" about="http://www.edwardbenson.com/"> Ted Benson </div></pre>
<p>(c) .. as CSS Microformats Equivalent data expressed using the vCard schema and conveyed as CSS Microformats.</p>	<pre><div class="vcard" rel="me"> <div class="n"> Ted Benson </div> </div></pre>
<p>(d) .. as HTML5 Microdata HTML5 Microdata provides a more explicit way to express data than Microformats but without mandatory URIs like RDFa.</p>	<pre><div itemid="http://www.edwardbenson.com/"> Ted Benson </div></pre>

Figure 2.2.: Various ways to add data semantics to HTML. At top is a set of RDF statements for reference. RDFa provides a way to intermingle these statements with an XHTML document. CSS Microformats are way to shim basic data semantics into CSS classes. And HTML5 Microdata is a more explicit way to express data objects within HTML but without the constraints imposed by RDFa.

In addition to custom extensibility, both XHTML and HTML5 support facilities for annotating data semantics atop the document object model. Figure 2.2 shows three different ways to annotate how an HTML page expresses data objects, along with an RDF fragment (using turtle [5] syntax) for comparison.

XHTML's support for data semantics comes from RDFa, an extension that sup-

2.2. Authoring and Reusing Web Documents

ports element attributes which represent the subjects, properties, and objects of an abstract RDF representation of the page [17] [45]. Attributes specify the subjects and properties of the RDF triples, and the content of HTML nodes specify the objects, as shown in Figure 2.2 (b).

Another effort called Microformats have advocated for an annotational layer based on CSS classes [55]. Much as standard ontologies offer a common vocabulary to describe common concepts, `microformats.org` contains a list of common semantic entities found on the web, such as contact information and products, and provides a list of CSS classes to demarcate them amidst an HTML document. The FOAF entity described as a Microformat would appear in HTML using the hCard vocabulary [20] shown in Figure 2.2 (c).

Finally, HTML5 offers its own mechanism for providing inline semantic annotation, called Microdata. More similar to RDFa than Microformats, Microdata provides attributes to specify item scopes, properties, and values but without the constraint that the participating data conform to the URI standards used by the semantic web. The contact information example using Microdata data is shown in Figure 2.2 (d).

These approaches are important because they show the power of (1) defining a standard for data markup and (2) letting others define their own vocabularies and schemas within that standard markup style. All of these standards have been put to great use by different communities. Microformats and RDFa, for example, are recognized by the web crawlers on many search engines. Standard vocabularies of data attributes are used to invoke widgets in many front-end web libraries like Twitter Bootstrap. And frameworks like Exhibit provide a standard language that others can copy, paste, and reuse without having to understand.

Cascading Tree Sheets builds on the strategy of microformats by allowing selector languages to associate relations with nodes. For HTML nodes, this can mean CSS classes, which enables CTS-based microformats that not only convey the existence of data but can also drive manipulations on that data.

2.2. Authoring and Reusing Web Documents

For authoring, direct editing tools help the user modify a web page with a WYSIWYG environment. These tools exist both online (such as CK Editor [58]) and offline

2. Related Work

(such as Adobe Dreamweaver).

A challenge with these tools that this work addresses is that they have little awareness of the semantics or provenance behind the structure of the web page they are editing. For example, targeting the CK Editor on the body element of a web document makes the entire page editable, in WYSIWYG fashion, by the user. For most modern web pages, there is not way to translate the edits that would result into meaningful data persistence operations: that web page may be an amalgamation of many database rows and template fragments, but none of that provenance is kept. To keep that provenance with today's tools requires extensive programming.

Chapter 6 will show that the lossless property of design rendering with CTS (Section D) preserves this provenance so that edits in the browser can be traced back to the proper data elements on the server responsible for those regions of the page.

2.2.1. Platforms for Reuse

Much literature has been devoted to study the phenomenon of design reuse. Scott Klemmer's work shows with designers shows that the practice of borrowing and tweaking the designs of others is commonplace design output can be improved by providing design galleries which facilitate this process [42] [62]. And Herring's work interviewing designers emphasized the importance of being able to easily publish, comment on, and search for design examples [46]. This thesis provides further data on the reuse practices of Exhibit authors, which provides further support for this work and data on how visualization frameworks in particular may facilitate community reuse.

Web Components

At time of writing, a proposal for encapsulation and reuse of custom HTML widgets is under development by the W3C and still at a Working Draft stage. This specification, called "Web Components," is actually four separate specifications, three of which have working drafts. While the details are sparse and the spec is still changing (i.e., a moving target for commentary) the proposal so far outlines three basic capabilities:

Custom HTML elements, which can be declared in Javascript with a new `register` method on the document or in HTML markup using the `<element name='my-new-element'>`

2.2. Authoring and Reusing Web Documents

element tag. The HTML that implements these element instances have their own private "shadow DOM" that is shielded from the rest of the page in certain ways, preventing, for example, CSS namespace collisions that can occur when widgets from different libraries are co-mingled on a page.

Design Templates, which are HTML nodes that wrap HTML content intended for use as a template. While these templates are parsed, embedded scripts are not run and images are not loaded. These templates can be used on their own or associated with a custom HTML element as implementation.

Web Imports, which allow HTML pages to import custom elements definitions--along with CSS, Javascript, and template dependencies--from other HTML pages on the web.

The Web Component standard was developed in parallel to the work in this thesis and is not fully supported by browsers at the time of writing, but it contains both many complementary ideas and also many important differences to the relational approach taken by CTS. Functionally, both extensions are a "webby" way to provide web templating and encapsulation of web content, and both assist in separating concerns between raw content HTML and design HTML.

A high-level difference is that of the focus of the two approaches. The Web Components spec appears to be targeted at strong encapsulation and easy invocation of web widgets. As such the composition model is not yet clear from the specifications (e.g., how a content author could specify a *particular* button widget for use inside an alert box widget), nor is the degree to which non-"scalar" mappings (i.e., an enumeration of items) can be mapped into a widget (e.g., a widget that displays a list of complex items). Finally, only HTML-to-HTML mappings are available.

By contrast, CTS is a more general-purpose language for content manipulation. Its bindings are more "loose" in the sense that CTS uses CSS selectors to refer to HTML nodes instead of providing a way to create new element names. Its composition model is clearly defined, but does not guarantee a private "shadow DOM" to prevent imported content from interacting with surrounding content. Finally CTS relations can be drawn, and the CTS engine can execute, upon any tree-shaped object, not just HTML. For example, in Chapter 6 we use CTS to connect and synchronize state between HTML nodes and spreadsheet objects.

2. Related Work

There is an authoring difference as well. Because `<template>` elements are not evaluated by the web browser, the production version of a Web Component can not be the same as the design artifact since, by definition, the browser will refuse to add the production artifact to the DOM and show it to the user. And the style in which Web Components are defined requires a component author to intentionally author and publish a component. Widgets written with CTS, by contrast, are plain HTML, so the design and production artifacts can be the same. Because CTS uses CSS selectors to bind content, and these selectors can address external documents, CTS widgets can even be crafted *post hoc* around an HTML page that was not created with the intention of reuse.

Despite these differences, the two approaches are born from the similar perspective that HTML is in need of a mechanism to provide better separation of concerns and reuse of assets. Given that Web Components are a W3C recommendation in the making, a useful lens through which to view this thesis is an investigation of the possibilities for a future extension or compliment to this new HTML feature.

XSLT

Extensible Stylesheet Language Transformations, or XSLT, is a Turing-complete [70][54] language designed to excel at transforming XML documents into other XML documents [23]. An XSLT processor takes one or more XML documents, along with an XSLT *stylesheets* that defines transformations on XML structures. The engine then walks the nodes on the primary XML input tree and uses XPath [22] pattern matching to find "templates" to apply to the current node to produce an output XML document. XSLT may be thus thought of as a general-purpose templating engine for hierarchical inputs (expressed as XML).

Cascading Tree Sheets is different from XSLT on several fronts. Semantically, the CTS model is also centered around symmetric relations between trees that express structural equivalences of various semantic roles (containers, enumerations, conditional existence, etc), much in the style of synchronous tree grammars [82]. These relations could be exploited to produce a variety of output trees depending on which tree is chosen as the primary tree from which to interpret the relations, and the engine which processes them is a simple top-down tree transducer (rather than a Turing machine). XSLT, on the other hand, is a declarative language for authoring one-way transformation functions.

2.2. Authoring and Reusing Web Documents

These basic language designs result in very different developer interfaces. XSLT requires expressing the output web page in terms of XSLT stylesheets—an XML programming language format. XSLT stylesheets can not be rendered meaningfully in a browser or HTML WYSIWYG editor because so many of the tags are programming directives. Ironically, because XHTML is also valid XML, XSLT Stylesheets that aim to produce XHTML as output must escape XHTML output in some circumstances so that it will not be erroneously interpreted as part of the XSLT document. Instead of the following fragment, for example:

```
<button onclick="append('<td></td>')">
```

An XSLT stylesheet would need to escape the embedded td tags:

```
<button onclick="append('&lt;td&lt;/td&gt;')">
```

CTS instead uses a mockup driven development approach, in which authors create normal HTML mockups and then associate relations with them from an external sheet, like CSS. HTML mockups in CTS can be loaded in a browser and edited in a WYSIWYG editor. The CTS rules that address them are kept separately and can be developed separately, creating both a physical separation of concerns between design and "programming" and a cognitive one.

Finally, XSLT's failure semantics are to crash on malformed input, showing no output at all if part of that output is ill-specified. The official HTML5 specification, in fact, permits the construction documents which XSLT would fail on, because HTML5 is not valid XML. While this is appropriate for enterprise-style programming, it is a questionable failure model for web development, which thrives on the kind of casual, copy-paste-customize programming that often results in minor errors. In fact, many developers knowingly load erroneous documents in the web browser precisely so that they can find and correct them. CTS adopts the CSS failure semantics, ignoring nonsensical input and reports failures in the console instead of crashing.

Automated Design Retargeting

Design retargeting is the practice of transferring the design of one page onto the content of another, something commonly done but until recently done only by hand until recent work on automating the process. CopyStyler [33] interactively helps

2. Related Work

users retarget entire pages with an interface that places them side-by-side. Fire-Crystal [71] enables a user to select an element of a web page and specify which visual characteristics to preserve. It then generates a fragment of HTML and CSS to reproduce exactly that style on any site. Bricolage [61] uses machine learning to perform page-level design retargeting automatically generating a minimum-cost edit script between two HTML trees.

For these semi- and fully-automated retargeting approaches, CTS provides both an annotational language with which to describe training data (in the case of machine learning) and also a format to describe the exact design transform being done so that it can be copied and reused without having to return to the tool responsible for finding that transformation. As with Polaris [83] and Wilkinson's grammar for graphics [93], there are great benefits to separating the runtime apparatus from the declarative language that describes the state of things.

2.3. Authoring Data Visualizations

Visualizing and browsing data is fundamentally different than text. A range of diverse skills, from data modeling and programming to design and aesthetics, are involved, and yet these skills are not often taught in conjunction with each other: people taught programming often aren't taught design, for example. As a result, the prior work on web visualization focuses primarily on providing *visualization capability* to a broad set of authors.

The Exhibit framework [48] provides web authors with the ability to write a simple HTML-based configuration to produce interactive data browsing interfaces complete with search, faceted navigation, and rich media types (such as maps and timelines). This framework has been used by several thousand authors in the wild and forms the basis of study and improvement for the data browsing chapter of this thesis. Because pages created with Exhibit are self-hosted by the authors that create them, it provides the ability to study the practices and struggles of end-users who seek to publish data, which Chapter 3 does. CTS draws upon many of the basic insights in Exhibit, namely that data-flow free HTML-based authoring environments are both usable by beginners but can be used to compose non-trivial user interfaces. CTS offers a generalized platform for such programming and also supports writing, whereas Exhibit is read-only and widget-based.

2.3. Authoring Data Visualizations

More common than the self-hosting style of Exhibit are data visualization *services* that provide hosting and embedding functionality for their users. ManyEyes [89] provides a variety of data visualization wizards for data sets uploaded to the site. Sense.us [44] collaborative hosted visualization authoring and annotation environment. The success of these sites is perhaps a testament that making these kinds of visualizations by hand is hard with the tools authors are familiar with. The packaging and reuse abilities of CTS can enable sites like ManyEyes to offer their functionality for dynamic *integration* with other sites without object embeds into a view pane controlled by a third-party application.

The Polaris software [83] [84], which later became Tableau [85], and Wilkinson's grammar of graphics [93] provide an good example of the value of creating specialized languages to describe a common domain. In their case, concepts like marks, scales, and projections are abstracted into a grammar that can describe the desired physical appearance of a plot. Once such a declarative language exists, different implementations of the engine that processes that grammar can be exchanged. Today with web programming, we lack a common language to describe the way data structures are combined, and the way changes to those data structures should be synchronized. Like these contributions to statistical graphs, CTS aims to provide a grammar for describing how web structures are related and can be composed and changed.

Other visualization environments are targeted at programmers rather than end-users. The Protovis [14] [43] and D3 [15] projects developed programming models that enable compact expression of a wide variety of complex visualizations. These environments require extensive programming, however, even for reuse. CTS provides a principled model for wrapping example programs written in these languages so that the example programs can be referenced, invoked, and re-parameterized from across the web. This could greatly increase the impact of these languages by enabling non-programmers to make use of them via simple application of CSS classes to a table of HTML data (see Section 5.2.2 for examples).

While all of these frameworks target authors from a range of skillsets, their evaluations focus on the language features and initial reactions from members of the populations they target. Absent from the literature is a follow-up evaluation of how these tools integrate into real-world authoring workflow. This thesis provides such a follow-up for the Exhibit framework and uses it to better understand both

2. *Related Work*

web visualization and HTML extension authoring.

2.4. End-User Front-end Language Design

Existing approaches to end-user language design form a spectrum of "flexible but difficult" to "easy but constrained". On one extreme are low-level procedural languages such as D3 [15] and Protovis. These frameworks can be applied to any visualization task but require significant skills and custom programming. Next are systems such as Scratch [73], Storytelling Alice [53] [38], and App Inventor [94], which still require procedural thought but offer a higher-level programming concepts and GUI to construct them. Yahoo! Pipes simplifies one step further by replacing the underlying programming language with declarative blocks that fit together to represent data flow and transformation [31] [49]. Finally are systems like ManyEyes [89] and Polaris/Tableau [83] [40] which provide turnkey visualization publishing but constrain user choice to a few fixed parameters.

The approach taken by both Cascading Tree Sheets and Exhibit occupy a place in this spectrum somewhere between Pipes and ManyEyes. This region is declarative and could be easily made GUI-driven. But unlike Pipes, it is free from any notion of data flow. And unlike ManyEyes or Tableau, it is still fundamentally a text-based and extensible (via Javascript) approach to visualization construction. The work in this thesis shows that this particular spot in the spectrum of programming environments strikes a compromise between usability and capability that meets the needs of a diverse set of end-users.

3. Case Study: The Exhibit Framework

The Exhibit framework [48] is an example of a web-based information management tool that is designed as a declarative extension to HTML rather than an application-layer library or tool. Exhibit allows anyone to publish an interactive structured data visualization by posting a data file and a single HTML document to the web. The HTML document contains special tags that describe both presentation and data interaction features, such as `<map>`, `<list>`, `<facet>`, and `<template>`. When the web page is loaded, Exhibit's Javascript library transforms these tags into an interactive display of any data files linked to the page. Since its introduction in 2008, Exhibit has been used by over 2,000 people to create visualizations like the ones in Figure 3.1 [7].

Exhibit's real-world deployment makes it a useful object of study to understand how the design of an HTML extension interacts with the community that authors HTML. Alternatives to Exhibit tend to be heavily programming-based (such as D3 [15], Protovis [14], and the many charting and graphic Javascript APIs) or heavily GUI-based (such as ManyEyes [89] and Tableau [85]). Neither of these two clusters involve ordinary HTML (sans programming) as input mediums. Both programming (even in Javascript) and GUIs use HTML as merely an output format, while Exhibit uses it as the medium of visualization authoring itself. CTS, as later chapter will show, provides a platform to adapt these other clusters for Exhibit-style authoring by end-users.

This chapter provides results of a mixed-method ethnography of Exhibit users [7] that examines usage of the framework on four primary levels:

1. What are the salient design choices in Exhibit that may serve a model for other HTML extensions?
2. Who uses Exhibit and why do they use it?
3. What kind of data and visualizations are made with Exhibit?

3. Case Study: The Exhibit Framework

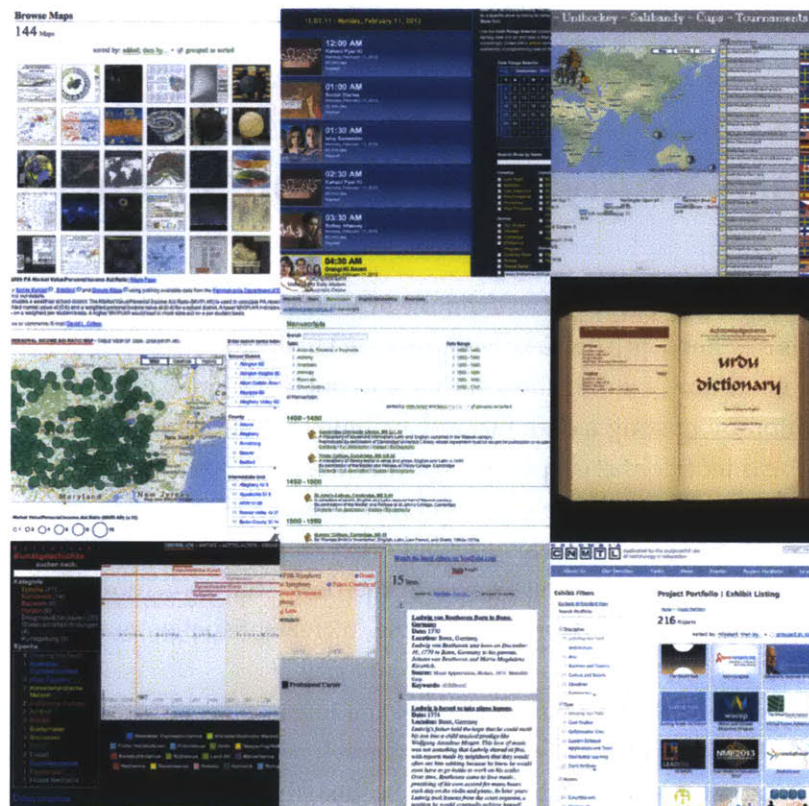


Figure 3.1.: Nine example Exhibits from our dataset. From left to right, top to bottom: a visualization of visualizations, a television programming guide, a floor hockey schedule, economic data, a library catalogue, a foreign language dictionary, an art history display, a Beethoven biography, and a curation of institutional projects. All of these pages were created with Exhibit using simple HTML and a dataset.

4. How does Exhibit authoring fit into today's web publication model?

The results of these inquiries are interpreted to provide insight to guide HTML extension development—how a framework's design affects ease of authoring and publishing. And from the human perspective, it characterizes the habits and capabilities of this community of long tail data publishers that authors in HTML rather than other tools.

3.1. **Methods and Dataset**

Exhibit lends itself well to quantitative usage study. Because Exhibit is web-based, any web page built with it can be scraped so long as it is published on the open web. The easiest way to use Exhibit is to link directly to the official copy hosted by MIT. The logs of this MIT server (which record the `http-referrer` header field) thus provide a way to locate Exhibit-based pages without having to scan the entire web.

Once scraped, an Exhibit-based web page can also be easily deconstructed into a standard set of components for analysis. Exhibit configurations are completely declarative, consisting of an extended set of HTML tags and attributes. These configurations, as well as the design decisions about the HTML page surrounding the visualization, can be isolated and compared across sites. A European Space Agency visualization and a floor hockey schedule (two Exhibits in our dataset) can be compared directly despite their wildly different data domains, for example. A declarative language combined with standard widget set further opens up the possibility of tracking user interaction behaviors across visualizations. Finally, the data that powers Exhibit visualizations can also be identified and scraped using information found in the visualization.

Using these methods, I curated three data sets for analysis:

The Viz1800 dataset is a list of 1,897 Exhibits located by processing the Apache server logs on the machine that hosts the Exhibit Javascript library. This dataset contains the complete set of assets used to create each visualization: the dataset, the HTML page surrounding the visualization, and the complete visualization configuration.

The Top100 dataset is a subset of Viz1800 corresponding to the 100 most visited exhibits according the Apache logs for a period of months in 2012. This dataset is used for analyses that required human coding, as well as to identify and recruit interview subjects.

The Interview dataset consists of recorded interviews with 12 Exhibit authors who responded to a request for participation sent out to authors we could identify among the Top100 dataset. They ranged from school administrators and teachers to business owners and urban planners. Each interview lasted roughly one hour and was performed either in person or over the phone depending on where the interviewee was located. Interviews were semi-structured, consisting of a standard set of

3. Case Study: The Exhibit Framework

prompts about visualization creation and maintenance, with follow-up questions based on the answers received.

3.2. Framework Design: the “Star Model”

Exhibit's design was originally built on the observation that many professional web pages (1) contained multiple views on top of a single dataset, such as maps, timelines, and tables, and (2) offered the ability to filter the data shown in those views with text search and faceted navigation. But these capabilities required custom programming, making them inaccessible to a wide audience. If these metaphors were simply added to the HTML tagset, this wider audience might more easily create these kinds of visualizations in their own pages.

The work published about Exhibit is grounded in the task of increasing the creative capacity of end-user web authors. But the choices made in the implementation of the framework turn out to be interesting as well, and they were largely undiscussed by the original paper. Exhibit's HTML vocabulary is just part of that implementation. The other part is the way Exhibit defines and constrains the interaction between these components this vocabulary invokes. These constraints enable a diverse set of applications while limiting authoring complexity.

The design pattern Exhibit uses to do this is worth naming and inspecting as a distinct point in overall space of end-user programming languages. It's shaped like a star, as seen in Figure 3.2, so we will call it the **star pattern**.

The star pattern describes a design in which all components an Exhibit user works with are independent given a data set at the “hub” to which they are all connected as spokes. Views, lenses, facets, and data sources are configured and added to an Exhibit without ever stating how they might interact, despite the fact that the entire purpose of, for example, a facet is to modify the state of a view. This independence is brokered automatically by Exhibit, which acts as a notification system for requests about, and modifications to, the data set at the hub.

This strong independence between components creates several restrictions on the types of user interface that can be created. The global filter applied to a dataset, pieced together from independent facets, must consist be either a strict conjunction or disjunction. Without dependence between filters there is no other sensible way to share or combine predicates. Components are also not able to configure the ap-

3.2. Framework Design: the ``Star Model``

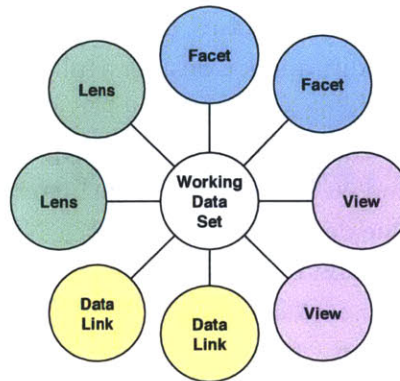


Figure 3.2.: In Exhibit's ``star pattern`` of design user components are configured independently of each other, and they only interact by manipulating or showing the data set at the center.

pearance of other components. And the class of applications that can be created is restricted to those in which the data displayed is a pure function of the parameters represented by the set of components.

But star pattern also enables a number of attractive usability properties. From the perspective of the author, the star pattern results in a framework that is:

- *Dataflow free.* All rules governing the sequence of decision-making, data transfer, and state update are automated by Exhibit's runtime library.

This eliminates scaffolding typically required in end-user programming systems like Yahoo Pipes! [31] [53]: understanding sequential computer code and describing how systems should interact. It also supports reusability: research projects have been entirely devoted to simplifying the task of reuse in sequential programs, which involves identifying the span of statements associated with some functionality and then adapting it to fit into some other set of statements [38]¹.

- *Connection Free.* If a single, global dataset hub is assumed in the absence of some other specification, then components may be added without any effort

¹This is not to denigrate either the practice of sequential end-user programming models or the work addressing their challenges---both are useful contributions. Rather, it serves as a useful contrast against which to compare Exhibit's dataflow-free design choice

3. Case Study: The Exhibit Framework

to connect them at all. In some cases, this supports wholesale reuse of components without any modification. Consider a full-text search box that filters the entire dataset. By copying and pasting this component from a visualization of political events to a television schedule, it begins to work correctly without any modification, and we know from design literature that copying and reuse is a critical part of the design process [62] [57]. Naturally, some components may have a configuration that is sensitive to the data domain (e.g., filtering based on only the `countryName` property), but this is an unavoidable problem in open-domain systems.

- *Isolated State.* While users may configure the state of a particular component, that state is isolated from the rest of the application and may be designed, copied, and pasted separately. If state were to be shared between components, as unconstrained frameworks allow, the upper bound on the complexity of the application parameterization would be k^n for n parameters shared across components (with range size k). By isolating each component from the standpoint of design, this complexity scales linearly with respect to new components (though still exponentially within a component).

The star pattern is not a framework itself, but rather a way to design and constrain the component model of a framework. As such, it could be layered on top of existing systems like D3 [15] to provide authors copy-paste-tweak semantics that enable them to invoke and reuse component functionality without worrying about data flow or widget implementation. Our interviews with Exhibit authors along with the diverse set of visualizations they created help validate the utility of this model.

Takeaway for CTS: The design of CTS will mirror these properties. While CTS' data model will consist of binary relations between nodes, these relation will be symmetric in their semantics, relieving the need of authors to describe the directionality of data flow. By enabling tree sheet authors to define a set of CSS classes that together form a CTS microformat, we can enable tree sheet users to invoke the functionality that tree sheet describes simply by applying those CSS classes---no explicit connections to imported components are necessary and the state of that invocation is isolated to the fragment of HTML with those classes. As will be shown, this is the basis for CTS' strategy for widget packaging and invocation, and in the long

run it could even be used to replace Exhibit's own view rendering system (while still relying on its data bus).

3.3. Exhibit Authors

Should new framework authors target end users? Or should they simply tailor new tools toward experienced programmers to increase the efficiency with which they can serve the needs of end users? There are many ways to approach this question, and a good start is to attempt to understand who the end-user web programming community is. This section contributes to the answer of that questions using author interviews conducted to find out who uses the Exhibit framework and why.

Exhibit offers a simpler authoring approach than programming-centric frameworks. But it still requires considerable effort on the part of its users: they must learn HTML, read documentation to learn an HTML-based configuration syntax, create and publish structured data, and publish raw HTML to the web. Understanding what skills they have and what tasks they struggle with can inform the design of other HTML-centric frameworks, and understanding why they went to the trouble of learning to use Exhibit when other "turnkey" systems exist sheds light on unmet needs in this community.

3.3.1. Author background and needs

Despite coming from the pool of the top 100 most visited Exhibits in our sample of 1900, none of the authors we interviewed were formally trained or employed in web development. Author 1's description is representative of the group at large: "I'm fairly comfortable with HTML. I've been doing HTML stuff for a long time. CSS is doable, but I don't feel as comfortable. I don't do too many fancy things, I guess." Subtle misuse of words from the technical perspective (such as describing HTML editing as "adding brackets") was common in over half the participants, indicating that most of this population is self taught. Three-fourths of the interviewees had prior experience editing HTML, and the other fourth encountered HTML for the first time when trying to use Exhibit.

Five of the twelve participants had prior experience with data visualization, however. Two used tools such as Tableau regularly, two were very familiar with cartography products, and one was the CEO of a consumer product website and was

3. Case Study: The Exhibit Framework

familiar with visualization from the business needs end. Three described themselves as having become advocates of structured data publishing, educating and providing support for their communities, and sharing new tricks they had learned about the tool. To apply Gantt and Nardi's *Gardeners and Gurus* metaphor [34], they were acting as local gardeners in their community.

Four common themes were present when authors were asked why they chose to use Exhibit to visualize their data:

(1) Data navigation. Eight of the interviewees mentioned wanting some way to let people navigate through the information they were publishing. For some, it was the primary reason they said they chose the framework. As an example, Author 3 saw the ability to dynamically filter and explore datasets as a competitive business advantage, both for customer-facing pages and his employees' own internal use. Author 11 and Author 9 both created their visualizations as a way to help themselves and coworkers interactively filter through sizable library catalogs that they managed for their departments.

(2) High-level data understanding. All of the authors who used maps or timelines, and some who did not, mentioned wanting to provide an understanding of their data in a different light than only text allowed. "I wanted to draw students' attention to different categories of information," Author 10, a professor, describes, "so having a marriage of these two visualizations [a map and a timeline] seemed very appealing to me." Author 2, also an educator, created curations of prior work for new projects. At first, "there were other people working on the project that said, 'Ah, this is great,' because they could have a quick look at all the different papers." Later, he discovered that the information displayed on facets themselves were a useful indicator of aggregate trends in his data (e.g., publications over time, or by school), not just a way to filter information.

(3) A programming-free way to publish data. Some interviewees were not thinking of themselves as data publishers, but rather just looking for a way to create a web page with information from some other source. Author 7 wanted to put an Excel spreadsheet of employee information into an HTML table. She had copied the Exhibit configuration from another department's web page and simply changed the data link. Unknowing that Exhibit provides faceted browsing and text search, she showed off the new employee-page they had recently paid a PHP developer to create that added facets to the employee directory, commenting that she wished "Ex-

hibit was capable of doing all this stuff [faceted browsing]" so that they wouldn't have needed a custom system.

(4) Frustration with traditional web development. Tech savvy interviewees also commented about their desire for respite from the complexity of web development. Author 3's company had gone through a custom PHP-based product catalog ("that was particularly clumsy"), storefront management software ("didn't work so well"), and Google's enterprise search solutions ("didn't really work well either"), before choosing Exhibit because it provides a product catalog with "fast, cross-referencable information" without any need to set up multiple web pages or maintain complex software.

These motivations affirm the initial needs Exhibit sought to target: navigating and visualizing sets of data without requiring programming. They also provide evidence that Exhibit's "star model" of programming provides a "pay-as-you-go" complexity that meets the needs of both novices (e.g. exporting a spreadsheet to HTML) and SaaS/hired software creation (e.g. product catalog).

Takeaway for CTS: There appears to be a useful median between no-holds-barred programming (i.e., PHP) and turnkey solutions. This median exposes authors to the structure of their data and interfaces in the same environment programmers would use (so they can extend if necessary, learning along the way) but without requiring programming to make something worthwhile. CTS similarly follows this model, exposing the author to CSS classes as binding points for pre-authored tree sheets, but offering a sliding scale of complexity. An intermediate CTS programmer might author their own tree sheets, referencing data structures elsewhere on the web. And an advanced CTS programmer might design tree sheets expressly for consumption by others.

3.3.2. Authoring Approach

Interviewees were asked to narrate how they designed and built the Exhibits in the Top100 dataset and any others they had constructed. Two consistent themes emerged from these stories: *trial and error editing* and *copy, paste, and tweak workflow*.

Trial and Error Editing Despite the lack of formal training, these authors were able to build an impressive array of visualizations, using HTML, and occasionally

3. Case Study: The Exhibit Framework

CSS and Javascript. The majority referenced a trial-and-error development style in which they would iteratively experiment with modifications to their code and check the results, often without firm knowledge of how to accomplish their goal. During one interview, an author whose HTML skills were among the most basic of the group opened the Chrome debugging console to demonstrate how he would experiment with different CSS styles.

This quick feedback was acting as a substitute for *a priori* knowledge, and the authors wanted more of it: a common feature request was to be able to edit Exhibit visualizations in the browser and save them to disk, as a way to further shorten this feedback loop.

A takeaway from this theme is that frameworks can support trial-and-error editing by offering a developer interface in which the average atomic unit of change between two working, but different, versions of software is small. Sometimes this property isn't possible. Making a change to a 3D videogame might require thousands of lines of code to be authored before any difference in application functionality is seen. On the other end of the spectrum, language like HTML, CSS, and Exhibit's HTML extension allow nearly every line of code to have a noticeably different impact on the program output.

Copy, Paste, and Tweak Workflow Armed with a basic understanding of HTML and the ability to experiment with trial and error, *code examples* to copy and modify were incredibly important to our interviewees. Nearly all interviewees reported creating visualizations by first copying an existing one and then modifying to suit their own dataset.

At one extreme was Author 7. She was an administrative assistant at a school tasked with creating a faculty directory for her department. Another department at her school was using Exhibit for the same purpose, and she created her visualization by copying the HTML code which configured Exhibit out of that visualization and changing only a single line: data link. She then used a tool to convert a spreadsheet of faculty into a JSON file.

Others made more in depth changes. Author 3, a repeat user of Exhibit, describes the value of having a wiki where other authors could post design examples:

I remember back. I was on that wiki a lot...The reason that the exhibits ended up being something I was able to use was that there were some

examples, and God I just wished there were more. It was one of the things I remember very clearly, going through and seeing how everybody did things. Examples were invaluable.

Author 8, who had made several Exhibits, said this of building upon his own work over time:

Basically taking an existing file, reusing that, editing it as need be, and then also...unless it's a real time crunch, I'm trying to learn more...add more features or add more components whether that's improved design on the CSS side or trying to do more with the data file.

This behavior is concordant with previous findings about both the practice and the utility of copying and modifying examples to create new artifacts [62] [42] [46]. A general conclusion that can be drawn from these experiences is that providing a space where the development community can curate examples can help end-user authoring a great deal.

A second conclusion can be made about framework design and the star pattern Exhibit employs, and that is that *ease of example creation* is an important characteristic of a language. Consider two hypothetical extremes to illustrate the point. In one framework, building code for reuse is an intentional act separate from other acts; production code is not inherently reusable. Content management system themes are an example in this direction: an HTML page that *looks like a blog* is not necessarily usable as a WordPress theme. To copy and then reuse a nice looking web page as a WordPress theme requires one to become a "theme developer," learning a specialized programming language, creating multiple files, and connecting to a database—despite the same underlying data schema. On the other hand, Exhibit widgets that share the same data schema can be copied and pasted from page to page without any modification.

Takeaway for CTS: Users are benefited by programming languages that allow users to peek inside an application, see how it works, borrow parts of it, and change it. The semantics of the CTS language, along with the lossless property of its rendering engine, will be shown to guarantee that web pages built with CTS retain all the relevant information that was used to compose them. If a visitor likes what they

3. Case Study: The Exhibit Framework

see, they can look at the web page source, copy it, and re-use the web template on their own page—even if the web template was rendered on the server.

3.4. Authoring Data

In this section, we also look to the kinds of data people visualize and how they create and publish that data to the web. We use both the Top100 and the Viz1800 dataset to understand the domain, scale, model, and format of data authored by the Exhibit community. Taken together with the interviews, the three chief conclusions supported by this section are: (1) Exhibit authors generally dislike—but often use—JSON as an editing format, (2) they curate data from a diverse range of domains which largely fits inside of spreadsheets, (3) they understand complex data models, but are stymied by lack of support for them in mainstream editors (like spreadsheets).

3.4.1. No love for JSON

All of the authors we spoke to authored data in a GUI-based tool, such as FileMaker Pro, Excel, or Google Spreadsheets. Even though Exhibit supports Excel as a data source, Excel-based authors reported converting their data to JSON for publication. After this initial export, some would later update the data in JSON format rather than its source format.

While a few authors described switching to JSON as an authoring format as they became more familiar with it, the majority had negative opinions about JSON's usability, citing it as one of the larger challenges of the visualization process. "I taught a course where we built [Exhibits] from JSON files, instead of Google Spreadsheets...and students unfamiliar with programming just found it too tedious and too difficult to chase after every missed comma and every unequal bracket...it defeated too many of them...quite a few just sort of threw up their hands in defeat," says Author 10. Author 3 has similar experience: "With JSON one of my concerns is not the coding but the missing brackets, and I've since found applications that help me find when I've got a missing bracket or an extra comma or something, but that gets to be an issue." As does Author 1: "that was the part that was less intuitive to me, setting up the JSON file."

These conversations suggest that the problem is one of syntax and tooling, not

understanding. These authors (and their students) are capable of organizing and thinking about structured data, but not skilled in the practice and tools of structured text editing that programmers take for granted. Editors that support syntax highlighting and bracket matching are largely unknown outside the technical community.

Takeaway for CTS: We will approach this problem by enabling CTS authors to use HTML as data format alternative to JSON and other structured formats. While HTML might not always be a sensible choice of data format (e.g., to encode genomic data), for many human-scale data sets it provides a nice set of tradeoffs. HTML can be viewed and edited via direct manipulation in a web browser, and it contains minimalistic display semantics for this scenario even without a spreadsheet (i.e., `<table>` elements appear as a table, `` elements appear as a bulleted list). If an author is making a table of floor hockey games to publish in a fancy visualization elsewhere, CTS will allow her to encode that data in what—in WYSIWYG mode—essentially appears as a rich text document but can also be used as a structured data source.

3.4.2. Diverse but Human-Scale Datasets

Put in spreadsheet terms, the average visualization created with Exhibit is 14 columns wide and 142 rows tall (with medians 12 and 67, respectively). The distribution of sizes is shown in Figure 3.3. The two are shown as a pair of histograms instead of a scatter plot because there is little correlation between schema size and item count ($r = 0.16$).

We visited each site in our Top100 dataset and performed three rounds of open coding to characterize the topic about which the data was created. This process resulted in the two-level breakdown of topics shown in Figure 3.4. As seen in the diagram, nearly half of the visualization in this dataset were devoted to the topic we called Information Resources, which consisted of abstract facts about the world (such a faceted list of chemical compounds) as well as references to other pieces of information (such as a list of research publications).

Interestingly, only ten of the twenty-three visualizations in the Events cluster made use of Exhibit's timeline visualization, despite "Events" being inherently temporal in nature. The other thirteen used only faceted text-based tables and lists

3. Case Study: The Exhibit Framework

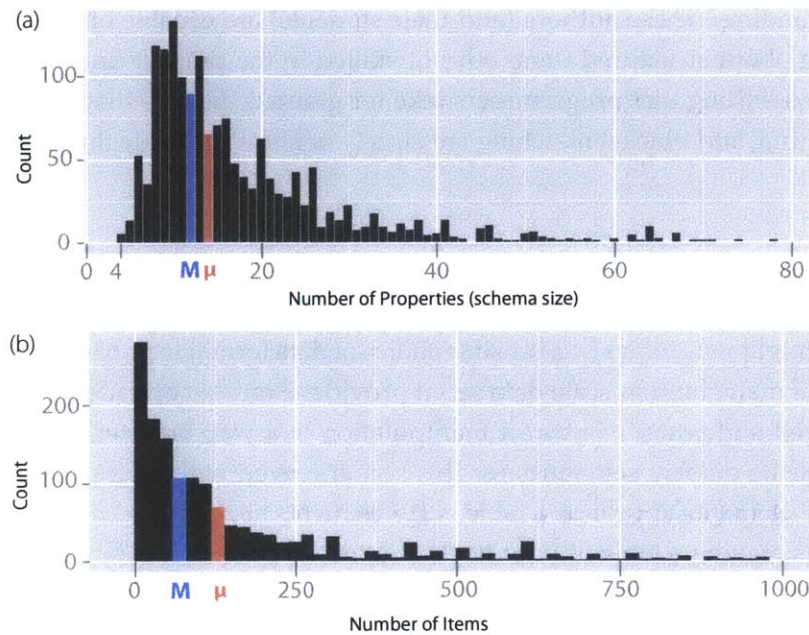


Figure 3.3.: **(a)** The average dataset used with Exhibit has a schema of 14 properties (median 12). This figure and calculation omits two outliers with more than 200 properties. **(b)** The average Exhibit has 142 distinct items (median 67). This graph and calculation is performed after clipping datasets greater than 1000, which hand-sampling shows are more likely to be generated by a dynamic database query.

could have easily been created with plain HTML except for the faceted data navigation. This supports the common refrain in interviews and prior work [51] that data *navigation*---rather than *visualization*---is more important for some authors. HTML alone does not suffice this need, as it provides only the ability to create tables and lists, but not to sort, facet, or filter them.

Takeaway for CTS: Here, we point to a more abstract takeaway, and that is the observation that people all sorts of data and visualization needs that do not fit easily into the workings of CMS software, which optimizes around display semantics (posts and pages) instead of bring-your-own-data semantics. While some CMS packages do allow custom data types (like Drupal), the editing interface they offer

42	Information resources	18	Research Papers		
		8	Facts (e.g., Chemistry, Municipie)		
		4	Library Resources		
		3	Reference Materials		
		3	Commentary (e.g., Blog)		
		3	Other		
		2	Projects		
		2	Datasets		
		27	People	18	Group Members
				4	Historical Figures
2	Groups				
2	Other				
23	Events	7	Historical		
		7	Classes		
		5	Other		
		4	Entertainment		
5	Objects	3	Maps, Books		
		2	Products		
3	Places	2	Brick and Mortar		
		1	Locales		

Figure 3.4.: Item types visualized in the hundred most popular Exhibits in our dataset. Types were and sub-types were refined using a three-round open coding process.

for them is not as direct as packing the data into a spreadsheet. Chapter 6 will explore basic content management functionality with CTS using only an HTML page and a spreadsheet.

3.4.3. (Ab)using Spreadsheets to author non-Tabular Data

A *data model* is an abstract set of rules that dictates how data can be structured. A spreadsheet, for example, uses a *tabular* model which represents a list of items (rows), each with the same set of scalar properties (columns). *Multi-valued tables* are like tables, but allow table cells to contain multiple values (e.g., a list of countries). And a *graph* represents a collection of nodes with relationships drawn between them. These abstract models have different breadths of expressiveness:

$$\text{Tables} \subset \text{M-V Tables} \subset \text{Acyc. Graphs} \subset \text{Cyclic Graphs}$$

Tools (and specialized languages) for authoring data are typically targeted at a particular type of abstract data model, such as spreadsheets (and CSV) for tabular data. But it is often possible to author data of any model using any tool as long as the writer and reader agree upon the rules to marshal and de-marshal it. Someone might author a multi-valued table inside a spreadsheet by placing comma-

3. Case Study: The Exhibit Framework

separated lists of items inside cells, for example, or they may author graphical data by agreeing upon the columns Subject, Predicate, and Object to record the graph.

We programatically analyzed each dataset in the Viz1800 collection to determine what underlying data model it used. To our surprise, less than half (41%) of the data sets were strictly tabular. 32% were multi-valued tables, 21% were acyclic graphs, and 6% were cyclic graphs.

Spreadsheets are the overwhelmingly predominant data authoring tool for end-users, and Exhibit authors are no different: all but one interviewee reported having authored their data in a spreadsheet application. Assuming the population of the Viz1800 dataset continues this trend, this breakdown of data model is alarming: most people author non-tabular data, but they are stuck using table-centric tools to do so.

While Exhibit provides workarounds for encoding nontabular data inside a tabular model (semicolon-separated lists for multi-valued tables and reference-able node IDs for graphs), spreadsheet UIs are not designed to provide any particular assistance for this kind of data. Spreadsheets do not allow authors to navigate or process lists encoded as semicolon-separated strings, for instance, and they force them into narrow cell layouts designed for numbers. Figure 3.5 shows that mismatches like this create enough friction to drive some graphical data authors to text formats (JSON) instead. A chi-squared test shows the distribution of publishing medium (JSON, RDF, or Google Spreadsheets) is significantly different for graphical data authors ($p < 2.2 \times 10^{-16}$, $df = 2$). Recall from our interviews that JSON was not a pleasant format among these authors, but they are driven to it by the lack of good graphical data support in spreadsheets.

This is a quantified gap between the observed habits of data authoring end-users and the tools the community has provided them. Either there has been a failure of industry to understand the sophistication of end-user data modeling, or a failure to properly integrate and educate authors about options available to them. The Related Worksheets line of work provides a promising way to enhance existing spreadsheet UIs to accommodate multi-valued tables and hierarchical data [4]. A similar way to enable better graphical data editing within spreadsheets would also be welcomed by the community.

3.5. Visualizations

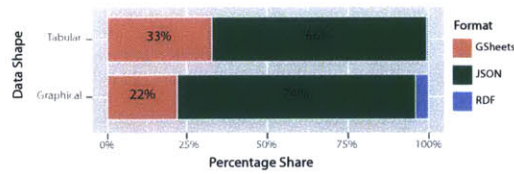


Figure 3.5.: Distribution of data authoring format used conditioned on the shape of data being authored.

3.5. Visualizations

This section examines how authors designed their visualizations. Recall that Exhibit visualizations are divided into three primary components: views, such as a map, timeline, or table; facets, which provide searching and filtering functionality; and lenses, which are miniature web templates for data items. These three components are woven into the rest of the web page with HTML.

Exhibits can have multiple views, displayed either modally or side-by-side. The mean view count is 1.4 and the median is one. 70% of Exhibits have one view, and 20% have two. The highest view count observed is 6. Exhibit visualizations have a mean of 3.1 facets and a median of 3. The most facets of any Exhibit in our dataset is 26. Only 17.5% of all Exhibit have no facets at all, which means that the vast majority of Exhibit authors both understand the concept of faceted navigation and consider the ability to filter sets of data a useful enough feature to include in their visualization.

This section uses the Viz1800 dataset to find three results. First, we identify three distinct data publishing needs: navigation, visualization, and “just plain publishing”. We show correlation between heavy use of a component and the complexity with which it is used. The most important and final result is that Exhibit authors tend to publish visualizations separately from text articles and need CMSs to allow them to take over the entire web page.

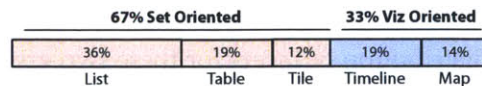


Figure 3.6.: View types used in single-view exhibits.

3. Case Study: The Exhibit Framework

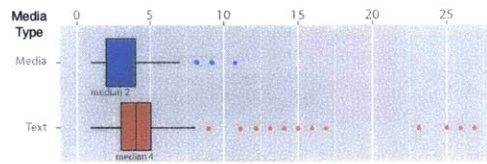


Figure 3.7.: Of Exhibits that have only one view, text-only views have significantly more facets than rich-media views.

3.5.1. Navigation, Visualization, and Just Plain Publishing

The motivation of the Exhibit paper hypothesized that there were communities of authors who wanted to (1) *provide navigation through* and (2) *visualize the relationships between* sets of items, but they were unable to do so because they lacked the proper tools. This claim was left unevaluated, and in this section we use data from the past eight years of Exhibit use in the wild to supplement our user interviews in demonstrating that this community, and these two needs do exist. We further show a third use evidenced in the data and interviews: using visualization frameworks as an on-ramp to web publishing for individuals not familiar with HTML.

To show this, we simplify the dataset by filtering it to only those Exhibits with a single view, 70% of all Exhibits total. This represents the set of visualizations with a focused purpose (i.e., just showing a map, or just showing a table), the breakdown of which is shown in Figure 3.6. 80% of these single-view Exhibits have facets.

Maps and timelines make up 33% of these Exhibits, which we claim are primarily used to visualize the relationship between data. Text-only Exhibits (lists, tables, and tiles) make up the other 67% of this collection. These Exhibits could have easily been reproduced in HTML alone *except for* Exhibit's faceting, sorting, and searching functionality. This extra functionality must have been the reason these authors spent time learning and deploying Exhibit on their site, demonstrating that data navigation is at least as important as visualization among Exhibit's user population.

By examining the distribution of facets for these two groups we can strengthen this distinction. We can infer that if an author's goal is to provide data navigation, she will have an incentive to provide more facets than if her goal was visualization of the relationship between items. Figure 3.7 shows a boxplot of facet count for the

3.6. Publishing

groups. Sure enough, text-only Exhibits (lists, tables, and tiles) contain significantly more facets than media-based Exhibits ($t(n = 1005) = -7.75, p < 2.2 \times 10^{-14}$).

Use of Exhibit as an on-ramp to HTML publishing is evidenced by the existence of text-based Exhibits with no facets at all. These tables or lists could easily have been ordinary HTML, but copying and modifying an example Exhibit configuration, which provides separation between the HTML design and data storage, must have been easier for these users than authoring the data as HTML. Author 7, who used Exhibit to publish a solitary HTML table, falls into this group.

Takeaway for CTS: Features that seem trivial from a web consumer's perspective—such as a sortable table—can be prohibitively complicated to implement in a web page for authors. CTS pursues a model that enables HTML convention alone to invoke interactive, Javascript-laden widgets and also to link in data sources to populate those widgets with data. While Exhibit offers a pre-made set of widgets, CTS develops a model to enable widget definition and parameterization by others.

3.6. Publishing

Authoring is only one small part of the overall process of digital creation. The publication step is the author's "last mile" to their work. This section explores the publishing experience and habits of Exhibit authors and synthesizes that into advice for framework designers. We save the takeaways for CTS design to the end of this section.

3.6.1. Platform Agnostic Client/Server Extensibility Needed

Most authors' web publishing experience is mediated by content management systems. But our interviews revealed that authors often felt held back by their CMS. The benefit of push-button publishing was had at the cost of constraining the kinds of artifacts that could be published.

Several authors we interviewed were able to create Exhibits on their own, but were unable to publish that Exhibit using their organization's CMS:

- "I wasn't able to publish the Exhibit myself. I had to go to the technology people and ask them put it up." [Author 1]

3. Case Study: The Exhibit Framework

- “I had to get help from a PHP developer. I’m not a PHP developer...to get help from an expert who knows PHP was a big plus.” [Author 7]
- “I have a love hate relationship with Content Management Systems...[they] very often have overhead that I don’t want to mess with.” [Author 10]
-

The crux of this problem seems to be that CMSs provide authors a very narrow canvas on which to draw: a contiguous rectangular region in the middle of the web page, with no easy access to side gutters or the HEAD element. Side gutters are important for design reasons—they are the canonical place for navigation—but the CMS generally does not permit page authors to modify them. And access to the HEAD element is sometimes necessary to include CSS and Javascript. Accessing either of these requires that the user become not only a system administrator, but also a programmer, editing low-level theme files. And authors who did this often made mistakes: our scrape of the web revealed several sites in which a theme file had been mis-configured to include Exhibit’s Javascript and CSS on every page, regardless of whether there was a visualization.

Despite the challenges of using external Javascript frameworks alongside a CMS, Author 5, who worked for a newspaper, believes that this is a better way to extend CMS functionality than native CMS plugin.

3.6.2. Publishing Environments a High Cost

The short story is this: Media organizations have serious technical debt. They have a CMS that is already costing them too much time and attention to support, and it is probably inflexible and closed source. This means media organizations can’t work with something new unless that something was explicitly designed to work with [the CMS] they already have.

There is an obvious workaround that every single modern CMS will allow you to embed custom Javascript into the published page. The ubiquity of this trait is why I can safely claim that front end technology [*author note: HTML, CSS, and Javascript*] is the most universal way to add modern features to a web experience in a legacy organization.

To avoid the problems we observed in our interviews, Javascript library authors should ensure their libraries can be included from within the authoring view of a CMS.

3.6.3. Exhibits need Wide Layouts

We coded the layout patterns of Exhibits found in the Top100 dataset. The results are shown in Figure 3.8, with the view area depicted in white and the facet/search areas depicted in red. The results show that left and right sides of a visualization are overwhelmingly the preferred places for navigational elements. Over 80% of all facets we coded were placed to the side of a visualization, and over 50% of visualizations used the sides exclusively.

CMS software typically constrains authoring to a center column of text flanked on the sides by one or two columns reserved for automated content (links, navigation, etc). But access to this center column alone does not provide authors the required width to display both a visualization and filtering widgets to manipulate the visualization. To support the visualization community, it is therefore important that CMS makers either allow authors to override the side columns with custom, page-specific content, or to take over the page entirely, stripping it of the CMS-provided UI chrome.

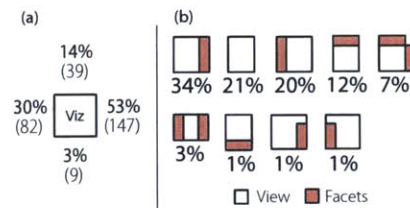


Figure 3.8.: **(a)** Raw counts of facet location in the Top100 Dataset. Most facets occur to the side of the visualization. **(b)** Counts of global visualization layout in Top100 Dataset. Over half utilize a two-column format.

3.6.4. Exhibits tend to be Stand-Alone

To examine how authors integrate visualizations with text content on their sites, we programmatically analyzed the HTML structure of the pages which contain each

3. Case Study: The Exhibit Framework

visualization in the Viz1800 dataset. For each page, we recorded two statistics: the total amount of text on the page in characters and the size of the largest single text node (e.g., what was the longest paragraph (<p>) element?). We also recorded these statistics for several well known websites. We then interpreted these measurements with the following heuristic: if a visualization is embedded in an article as supplementary information, we should expect the text statistics of the web page to appear like other pages known to contain articles. If the visualization stands alone, on a page by it self, we should expect much less text than on an article-containing page.

Figure 3.9 shows a scatterplot of the data. The horizontal axis is the amount of total text on each page, and the vertical axis is the largest block of text. Web pages containing Exhibits are shown in black, and reference points are shown in red: the 2014 CHI CFP, the front page of Slashdot.org and Reddit.com, and randomly selected articles from the New York Times, Washington Post, and Chronicle of Higher Education.

These results show that Exhibit visualizations are overwhelmingly published as stand-alone content that occupies its own page, not supplementary figures woven into articles. 85% of the points are strictly bound by the point representing the New York Times article, itself the smallest of our reference points. The remaining reference points are at the extremes of the collection. The line visible at $y = x$ represents web pages that contain only a single unit of text outside the visualization.

CMS builders can use this information by prioritizing support for page-centric visualizations over interactive features embedded in prose. It may be that we simply have not yet developed the journalistic culture of mixing the two forms yet, though recent publications like the Snowfall feature by the New York Times [16] and several interactive web adaptations of papers from Berkeley's AMP lab [59] [3] show that there is high payoff when text and interactivity are integrated well.

Takeaway for CTS: Content Management Systems are both incredibly enabling (for the common case) and incredibly constraining (for the long tail). Many CMS constraints that frustrate authors trying to publish custom data and visualizations are rooted in fact that the functionality of the CMS is described by procedural and relatively unchangeable code. This thesis will show that many of the core functionalities of the CMS, such as theming (Chapter 5.2.4) and read-write-update interfaces (Chapter 6) can be built into the web at its core using a relational layer, removing much of the need for large, rigid software stacks. If a language like CTS were

3.6. Publishing

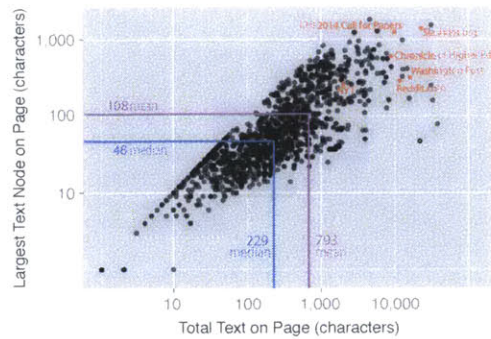


Figure 3.9.: Nearly all the Exhibits in our dataset are stand-alone visualizations, not inline supplements to an article. Reference points (red) of articles on the web contain far more text (horizontal axis) and far larger paragraphs (vertical axis) than the pages in our dataset.

adopted more widely, CMS software would no longer have to manage the nuts and bolts of theming and saving content edits back to the server. Instead, it could focus on providing authors with a flexible creative authoring environment that met the needs of both the long tail and the common case.

4. Cascading Tree Sheets

The web development ecosystem thrives at the application level, with frequent appearance of new programming frameworks, code preprocessors, and widget libraries. But the underlying declarative stack on which these libraries is based has been comparatively stable. Both HTML and CSS have received significant overhauls over the years, but with a few exceptions, these improvements only extended the existing vocabulary of these two languages without broadening the *kinds of statements* that web sites can make about data.

If the web community were to add a new, third declarative language to the web stack, what kinds of statements would it make? How should it integrate with the web development process and existing web content? And how might it benefit web authoring and development?

This chapter presents Cascading Tree Sheets (CTS), a prototype language created to explore these questions. CTS is a "CSS for structure," allowing authors to make statements about how structures on the web relate to other structures on the web.

The hypothesis behind Cascading Tree Sheets is that *relationships between structures* is a category of statement that is missing from HTML and CSS and that would yield great benefit for authors and developers. After all, most of the application code programmers write for the web is involved in ferrying data around from place to place: from database to web server, from web server to browser Javascript, and from Javascript to HTML. A layer which describes the relationships between all of these sources and destinations might alleviate many of the problems related to these tasks. This relational layer call this the *CTS model*.

The *CTS language* layered on top of that model is a set of five particular relationship types implemented in the CTS prototype. We show how these relationship types can be interpreted as sufficient information to achieve a broad range of tasks common to web applications today, such as templating, scraping, reactive UIs, WYSIWYG editing, light-weight content management, and fine-grained caching in the browser. We'll also show how the user interface of this language is CSS-like,

4. *Cascading Tree Sheets*

enabling a variety of interesting reuse scenarios.

The goal of this chapter is to get you thinking about CTS and how it might impact web design. The design of CTS is the kernel of innovation in this thesis, but the resulting impacts (detailed in the following chapters) are the punchline. For this reason, this chapter will attempt to acquaint you with CTS as a user. Just enough to understand what is going on, but (hopefully) without belaboring implementation details and proofs. For those details, please see Appendix B for the execution model, Appendix A for the formal grammar, Appendix D for a further details about CTS' lossless rendering property, and Appendix C for examples of the internal rendering state under different examples.

So without further ado, let's talk about CTS.

4.1. An Intuitive Introduction

Let's say you are building your own website. You created a basic, no frills HTML page with the information you wanted to put on your website. Your designer friend made you a wonderful design mockup based on that simple page you gave him. You can't just publish this design mockup, even though your friend took the liberty of loading it with your demo data for you, because you want the website to change over time: you might post new announcements, photos, etc. How would you go about the task of building a system that lets you update data in one place and have it appear in the finished, stylized page?

Try to unlearn everything you know about web programming and web templating for a moment, and imagine what this process might look like starting from a fresh slate. If you printed the two documents and were showing them to someone in a coffee shop, how would show them how they are connected?

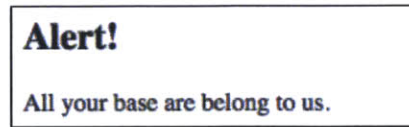
You might start alternating between the two documents, pointing different bits out and saying, "See this thing here? That's the same as this one over here." If you were to say that, you would have just written your first line of code in CTS.

Let's look at a specific example. Figure 4.1 shows a simple version of this scenario. At left is a design from your designer friend (whether this design is actually fancy is left as an exercise for the reader). At right is the simple page with the data you want to put on that page and update from time to time.

Again, thinking about explaining how the two fit together on a paper printout,

Designer Friend's Fancy Design

```
<div class="panel panel-primary">
  <div class="panel-heading">
    <h3 class="panel-title">Panel title</h3>
  </div>
  <div class="panel-body">
    Panel content
  </div>
</div>
```

Your Simple Page

```
<h2>Alert!</h2>
<p>All your base are belong to us.</p>
```

Figure 4.1.: An example of a web design and the content that might be destined for integration with that design.

we might say,

1. "The panel title and the Alert go together"
2. "The panel content and the 'all your base' sentence go together"

Cascading Tree Sheets lets you make the same sorts of statements. In order to point out what things are, we will use *selectors*: CSS selectors if we're pointing out a bit of HTML and some other selector language if we're pointing out a bit of a different data type. Practically, these "things" we're selecting are going to be tree nodes, e.g., HTML nodes in the case of Figure 4.1. We can name the trees we're pointing at so that we know which selector goes with which tree. Finally, in order to state how the two items are related, we're going to create *symmetric binary relations* between the items selected, a natural way to describe relationships that we will justify a bit more later.

Here are the actual CTS statements to do just this:

```
FancyDesign | h3.panel-title :is SimplePage | h2 ;
```

```
FancyDesign | div.panel-body :is SimplePage | p ;
```

In the above code, `FancyDesign` and `SimplePage` are the names of the two trees, `:is` is the relation, and the rest are the selectors.

4. Cascading Tree Sheets

What might we do with these relations once we've made them?

- We might use them to take the data from the simple page and put it in the fancy page, and we could call this **web templating**
- We might use them to scrape the data from the fancy page out into the simple page, and we could call this **web scraping**
- We might use this to synchronize data back and forth as the user edits the page live on the site. When they edit it in one document, we update the other document. We could call this by the popular term for it at time of writing, **reactive programming**.
- We might automatically generate a user interface on top of either web page, offering UI affordances which allow a user to invoke an editing interface to change the title or to change the message. We would change the nature of the editing interface based on the kinds of relations that we observed on the page. We could call this a lightweight **content management system**.
- Given that we can do all the above, we might stop calling the fancy page a "page" and start calling it a "theme," because anyone with the two statements above can import and reuse it. We might develop other themes that work with the same two statements so that we could swap as desired, and we could call this **theming**.

Of course, this simple `is` relation can not handle every situation that comes up. What if the design had lists of items, or optional items, or needed to invoke yet other designs itself? Intuitively, we would need to build up a set of relations that work together to handle these scenarios in the same way that template languages have sets of keywords for producing HTML content.

This chapter will develop all these concepts a bit more fully, into the language called Cascading Tree Sheets. But despite all the details that will come into play, the basic idea is nothing more than this scenario.

Before moving on, let's briefly address the immediate question that should arise about whether what we're about to develop in this chapter is "just another template language" or "just another reactive programming framework". Cascading Tree Sheets can template your content, and it can also create a reactive interface, but those are just **two applications** of what's really going on: the introduction of

a new layer of information on top of web documents that lets us understand how they are related, reason about how they fit together, and reason about how to act on that information. Like CSS, this new layer of information is both physically and logically separated from HTML. It annotates from afar, allowing HTML stay as it is in mockup form, without interleaving it with other languages. The core innovation is this new layer of annotation, its design, and its properties. To get the most out of this thesis, think of Cascading Tree Sheets as a materialization of this core idea rather than a template language or reactive programming framework.

We now rewind a bit and revisit this story taking more time to unpack the particular design of CTS. In turn, we will address its model, its surface form, and its semantics. Then we provide examples and revisit template rendering.

4.2. Model

The abstract data model of Cascading Tree Sheets is composed of three core concepts: trees, relations, and events. All web resources addressable with CTS are viewed as *trees*. These trees are said to live in the same *forrest*. Based on the selectors found in CTS statements, *relations* are formed between trees in the forest. When the structure of one of these tree changes, *event* objects thrown and passed from node to node.

This section explains the specifics of the underlying data model of CTS trees, relations, and events. It also details the particular tree structure used to represent the three data types the CTS prototype supports: HTML, JSON, and Google Spreadsheets. While this may seem like an implementation detail, it is a design choice with major implications for how the user interacts with these trees. For example, addressing a spreadsheet as a table of items and addressing an HTML attribute are both issues related to the way in which CTS proxies these two data structures.

4.2.1. Trees

Trees are the organizing structure addressed by CTS. This is a convenient abstraction for a number of reasons, first and foremost that HTML is a tree-shaped data structure. But all two-dimensional user interfaces are also, by nature, expressed hierarchically and often thought of as tree-shaped data structures. And graphical

4. Cascading Tree Sheets

and relational data is easily—in fact inevitably—projected as a tree for display to the user.

Cascading Tree Sheets can thus address and manipulate any data object published to the web so long as the CTS engine understand how to interpret that data object as a tree, and then translate manipulations to that tree back into operations on the data object.

From the implementation perspective, nodes in CTS trees are objects deriving from a common base class that provides convenience mechanisms such as event handling and provenance tracking. Particular tree types, e.g. HTML or JSON, may subclass this base class to provide extra functionality necessary to manage handling the data structure that tree represents.

From the data model perspective, nodes in CTS trees labeled with a 2-tuple defining the node type and an optional value. For example, the HTML element `<p>` would be represented by a node with the label `<HTMLElement, P>`. Or a node representing the spreadsheet cell A1 might have the label `<Cell, A1>`.

Casting data objects as a tree often results in a structure slightly different than the original, even when the original was a tree itself. The DOM Tree which represents an HTML page, for example, must be expanded to allow addressing attributes and text nodes in a more homogeneous fashion than the DOM API exposes. In some cases, this reinterpretation requires that data be projected redundantly along different dimensions, as is the case with spreadsheets (Chapter 6). While such redundant projections require a bit of state coordination, requiring that every data object assume tree shape enables otherwise disparate data resources to become easily merged.

HTML Trees

The tree representation used for HTML largely mirrors the browser's DOM representation with a few small changes related to attributes and children. A DOM node in the browser has both attributes and children, but CTS nodes have no notion of attributes—only children—so the CTS representation of HTML must treat attributes as children. To do this while still maintaining separate notions of attributes and children, CTS HTML nodes lazily maintain two special, fixed children: an *E* node, which represents the child container, and an *A* node, which represents the attribute container. Figure 4.2 shows an example HTML fragment along a diagram showing

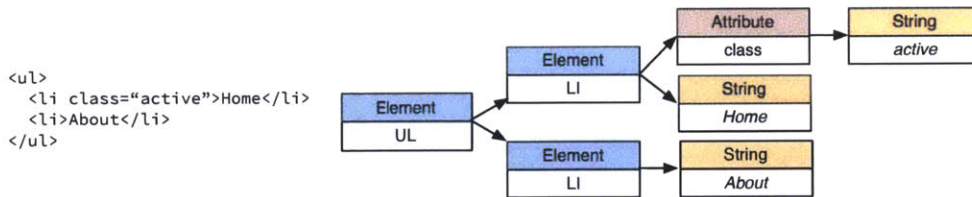


Figure 4.2.: An example depicting how CTS interprets HTML as a tree. Because CTS does not have any notion of node attributes, HTML attributes are represented as tree nodes and placed in a special attribute container underneath each Element node.

corresponding CTS tree structure using this *E/A* distinction.

CSS selectors are used to select nodes from HTML trees, but we adapt these selectors to accommodate attributes.

1. To select an Element node, use a CSS selector that would ordinarily correspond to this node. For example, the selector `li.active` selects the first `li` node in Figure 4.2.
2. To select an Attribute node for the attribute *X*, append the key-value `{"attribute": "X"}` to a selector which selects the corresponding Element node. For example, the selector `li.active {"attribute": "class"}` selects the class attribute of the first `li` node in Figure 4.2.
3. To select all attributes of a node as a collection, append key-value `{"attribute": "*"}` to a selector which selects the corresponding Element node. For example, the selector `li.active {"attribute": "*"}` selects the *A* node (representing the attribute collection) of the first `li` node in Figure 4.2.
4. By default, when a relation addresses an element node as a collection, the collection used is the *E* node unless the attribute collection was explicitly specified in the selector which cast the relation.
5. There is no way to select nodes that represent primitive values, such as the String nodes in Figure 4.2.

4. Cascading Tree Sheets

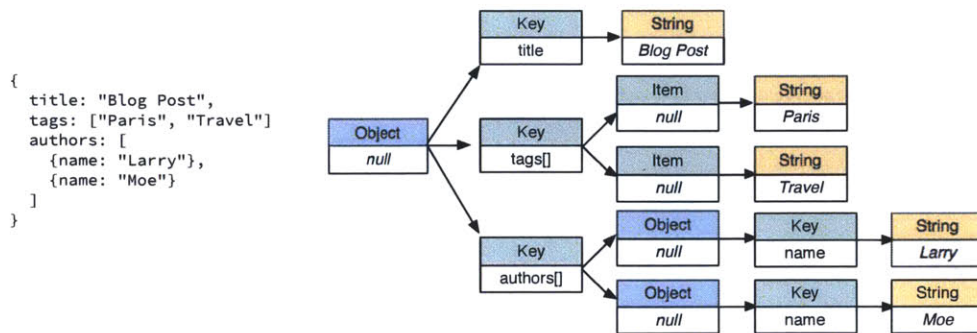


Figure 4.3.: An example depicting how CTS interprets JSON as a tree.

JSON Trees

The tree representation used for JSON mirrors JSON's natural structure but represents objects such as arrays and dictionaries using tree nodes:

- **Object** nodes represent JSON objects. Its children must be only **Key** nodes, and its value is **null**.
- **Key** nodes represent keys on a JSON object. Their value represent the name of the key.
- **Array** nodes represent a JSON array. Its children must be either **Item**, **Object**, or **Array** nodes, and its value is **null**.
- **Item** nodes are wrappers around primitive value nodes. They exist only to act as containers to support lossless rendering by the CTS Engine (see Section D for details).

An example of this structure is shown in Figure 4.3.

Both **Key** and **Array** nodes may have nodes representing primitive types as their children.

One requirement CTS imposes on JSON trees is that they are rooted in a dictionary. This simplifies the JSON Tree's selector language by ensuring that all primitive types have at least a string label (the dictionary key) that refers to them.

JSON trees use a slightly modified version of JSON's standard dot-path notation. A wildcard character (*) can be used to reference all children of an **Array** node, in

which case the selection proceeds into all children of that array. For example, in Figure 4.3:

- The selector `title` would select the `Key` node with value `title`.
- The selector `tags` would select the `Array` node that is the child of the `Key` node with value `tags`.
- The selector `tags.*` would select all `Item` nodes of the `Array` node that is the child of the `Key` node with value `tags`
- The selector `authors.*.name` would select all `Key` nodes with the value `name` in the figure.

4.2.2. Relations

Having represented various web objects as tree structures, the second component of the CTS abstract model is relations between the nodes that make up those structures. Building up the CTS language using relations as the fundamental building block makes sense given the language's broader purpose. Consider the narrative that accompanies design and content organization within web data: "*this thing and that thing go together; these bits over here are joined to those over there*" and so on. These kinds of statements are relational in nature, and the CTS language aims to encode them.

CTS relations are binary and symmetric. Each relation may be written as $r(n_1, n_2)$, where r is the relation type and n_1 and n_2 are tree nodes. Because of symmetry, this is the same as stating $r(n_2, n_1)$. As will unfold during a discussion of the CTS execution model, symmetry is a key property that enables CTS to automate the implementation of reactive data interfaces.

4.3. Surface Form

So far in this chapter, we have established the high-level motivation for Cascading Tree Sheets: encapsulating and reusing structure on the web. We have also established that its data model will be symmetric binary relations. Before delving into particular statements the CTS language might enable (see Semantics, Section 4.4) this chapter will address the the *surface form* of CTS: the way a CTS developer will

4. Cascading Tree Sheets

author these symmetric binary relations. This way, when it comes time to discuss semantics, we will have already laid the groundwork for authoring CTS statements that we can point at them and talk about them.

To develop a surface form for expressing these concepts, we look to the success of CSS as a language for making comments about HTML documents. CSS offers an extraordinary flexible range of relationship possibilities vis-a-vis the HTML it accompanies. CSS statements can be inlined on a single HTML node or externalized in a separate file for application to many HTML nodes. They can be authored to fit existing HTML, or authored with the intention of informing the design of future HTML. And statements can be associated with HTML nodes based on their structural context or their semantic role (via classes).

The developer-facing component of the CTS data model is symmetric reified binary relations. Relations are crafted between nodes, and nodes exist in trees. In order for developers to author CTS, they must therefore have a way to reference *trees* and *nodes*, and to specify *relations types*. More casually, back to the original example, authors need to a way to say: *“this thing here goes with that thing there.”* Expressed like that, a link to CSS is immediately apparent: CSS selectors provide an excellent way to define notions of “this thing here” and “that thing there” on a web page.

This section applies this root observation to create a CSS-like surface form for the CTS language. The aims for a description just specific enough to allow a reader to read and interpret the CTS examples in this thesis and reason about designing web applications with the CTS language. For a precise description of syntax, see Appendix A.

4.3.1. CTS Statements

We will build on this notion of selectors to create the surface form that developers will use to author CTS relations. A single CTS statement will be of the following form:

```
Selector1 Relation Selector2 ;
```

In the prior section on the CTS data model (Section 4.2), each tree type supported by the CTS prototype was paired with a quick tour of the selector language CTS uses for that tree type. While the reason for these selector languages should now

be apparent, their particulars are, for the most part, an implementation detail. The important pieces to consider are:

- Each tree type CTS works with has a selector language, and
- Selectors in the language are queries that can be evaluated against the tree to provide a list of zero or more matching nodes

Next we will need a way to talk about trees. For this, we will define a notion of a CTS header, modeled after the At-Rules in the CSS specification:

```
@ResourceType ResourceName ResourceUrl ;
```

Taken together, we can rewrite the archetypal CTS sentence as:

```
@html OverHere http://here.com/here.html ;  
@html OverThere http://there.com/there.html ;  
  
OverHere | ThisThing :GoesWith OverThere | ThatThing ;
```

4.3.2. Tree Sheets, Style Blocks, Inline, and Imports

Just like CSS, CTS may be attached to a web page in four different ways.

Tree Sheets like style sheets are separate files containing CTS statements. They may be published to the web and included in a page using an HTML link tag, just like style sheets.

Style Blocks are tree sheets embedded directly inside an HTML document using a `<style type="text/cts">` tag. Their form is otherwise the same as a linked tree sheet.

Inline CTS statements are placed within the `data-cts` attribute of an HTML node. In this case, one of the two selectors must be the special selector `this`, which is a query that always resolves to the node on which that CTS statement is attached. For example, an author might write:

```
<div data-cts="this :GoesWith ThatThing;">
```

4. Cascading Tree Sheets

Otherwise ordinary CTS statements are valid, albeit subject to the syntactic requirements of HTML attributes.

Import Finally, it is sometimes convenient to allow a tree sheet to import other tree sheets. We permit this using the same kind of header statement we use to include trees. The `@cts` header statement will cause the CTS engine to import the referenced file as if it were part of the current tree sheet. For example, an author might write the following header in a tree sheet:

```
@html OverHere http://here.com/here.html ;
@html OverThere http://there.com/there.html ;

@cts http://path.to/some/other/treesheet.cts ;
```

4.3.3. Statements versus Relations

An important point to reflect on is the difference between CTS statements and the relations that they ultimately are used to instantiate. Authoring selectors (queries) rather than relations creates an important layer of indirection that enables reuse.

Consider the example of CSS Microformats [55]. Microformats are perhaps the canonical example of an attempt to standardize common sets of CSS classes to represent certain semantic entities on page, such as contact information or calendar items. Wherever these bits of information occur, if the HTML author applies those CSS classes to the right HTML nodes, it signals to a microformat data scraper that data entities exist and can be extracted from the page.

Similarly, basing the CTS surface form on selectors, not relations, allows for curation of sets of CSS classes to represent *some CTS-based operation* (that we haven't defined yet!) for reuse across sites. Just as one does not need to be a CSS author to participate in a CSS microformat, in this scenario one would not have to be a CTS author to participate in a CTS microformat. Simply applying the correct set of CTS classes and linking to a tree sheet would be sufficient. We will explore this idea as it relates to *post hoc* site theming in Section 5.2.4.

4.3.4. Default Trees and Late-Binding Trees

While the CTS language as we have defined it thus far could apply to any tree structure, the surface form is obviously biased by the fact that it is intended for use in HTML documents. Continuing with this special treatment of HTML, we define the notion of a *default tree*, which is whatever HTML document occupies the position of the global document object in Javascript's context. If a CTS statement does not have a tree reference in a selector, that selector is presumed to be targeted at this default tree.

It also important to point out that tree references can be made without binding those reference to a tree. Consider the following hypothetical scenario: a skilled CTS author uses our yet-defined language to combine an online weather map graphic with some JSON data from the national weather service. The result is a nice depiction of the weather for that day.

This author could publish that tree sheet for reuse by binding all selectors into the JSON data object to the tree name `weatherData` and then *leaving out* the particular header statement which binds the tree name `weatherData` to a particular URL.

Someone who wanted to reuse this author's creation with different data (for a different date or from a different source, perhaps) would only need to bind that tree name to the new source. This relies on an assumption, of course, that the selectors also work for the new data source.

While the CTS pro may have authored a complicated tree sheet, the CTS novice who is reusing the work would have a tree sheet that looks like the following:

```
@json weatherData http://path.to/my/different/dataSource.json
@cts http://path.to/fancyMap.cts
```

4.4. Semantics

Now that we have hammered down the model and surface form of the CTS language we can begin to address semantics. Most programming languages have a set of keywords that define the most atomic semantic statements a programmer can make. Since the CTS language is composed of relations between trees, CTS has a set of *key relations* instead of keywords. They are: `is`, `graft`, `are`, `if-exist`, and `if-nexist`.

4. Cascading Tree Sheets

This section addresses the semantics of these key relations and provides examples of when and why a CTS programmer might use them. We begin by motivating the choice of relations by looking to the functionality that template languages provide. We then explain each relation by transferring concepts from the template language world onto CTS' relational model.

4.4.1. Choosing what concepts to represent

If the goal of CTS is to create a prototype language that helps wrangle web content, a good place to look when defining the semantics of that language is template languages. Template languages are the tool through which software developers encapsulate bits of HTML design and weave these bits together with data. If a new declarative language for wrangling web content is to be proposed at the root of the web stack, at the very least it seems reasonable that it should be capable of describing the kinds of operations we use template languages to perform today.

As often implemented today, template languages may be thought of as an inverted code file. In typical programming, a line of text represents program code and output statements need to be specially escaped and wrapped in some `print` command. In template languages, a line of text represents a program output, and it is *programming code* that needs to be specially escaped and wrapped in some sort of delimiter.

Whether the template language is procedural (such as PHP [72] or Ruby's ERB [65]) or declarative (such as Handlebars [52] or Velocity [1]), or whether it is string-based (most languages today) or tree-based, most popular template languages today can be thought of as computer programs that take an input and produce an output. The task for designing CTS, then, is to determine what sort of control flow is present in these domain specific programming languages so that we have an empirical understanding of what might be useful for CTS to describe.

Table 4.1 shows a survey of web template languages done to help with this needfinding. While different template languages offer different functionality, it appears that a common denominator are the following four concepts:

- **Print (Including Data).** The most basic functionality of template languages is to transclude data into a design. In other words, the statement `print(x)`.
- **Include (Calling other templates).** Calling other templates enables a design

4. Cascading Tree Sheets

to be modularized and broken into separate files. For example, the header of a web site may be stored in a `_header.php` file and included on many different pages. These sub-templates can be parameterized, though proper support for encapsulation varies from language to language (e.g., the sub-template may simply adopt the scope of the caller).

- **If/Else.** Some form of conditionality is useful so that the visibility of design elements can be toggled based on dynamic data. Of the three languages that do not support arbitrary boolean conditions for an if/else block, two of them at least have an existential if/else instead.
- **Foreach.** Looping over a set of items in order to apply a common design to each item is the final piece of functionality in common to all these template languages.

4.4.2. Fitting Template Operations onto a Relational Model

We now adapt each of these four basic template operations—print, include, if/else, and foreach—for use with the symmetric binary relations that CTS employs. This process is not a direct mapping for two primary reasons worth pointing out here.

First, template languages are procedural: they specify an ordered script for transforming inputs into an output. CTS on the other hand is declarative. The ordering of CTS statements (as written) do not matter, and CTS relations merely state relationships between nodes rather than prescribe a set of actions.

Second, traditional template commands have only one argument list (e.g., `print(x)`) and assume the context in which that printing occurs from the document. But a symmetric binary relation representing the same concept could be interpreted in two different ways. For example consider a symmetric relation called `print`. Because of its symmetry, `print(x,y)` could either mean object x should be transcluded into object y or vice versa. Expanded to cover the whole set of functionality, this means two related documents could be processed by a relational template engine in either direction.

The five relations CTS uses which represent the translation of these template concepts are:

- **is**, the analog of `print`, which denotes atomic content equivalence between two nodes

- `graft`, the analog of `include`, which denotes a structured content equivalence two nodes
- `are`, the analog of `foreach`, which denotes set equivalence between two nodes
- `if-exist` and `if-nexist`, the analog of `if exist` and the paired `else`, which denote two types of existential dependency between nodes

We will now explain each in more detail and provide examples.

Is — Atomic Content Equivalence

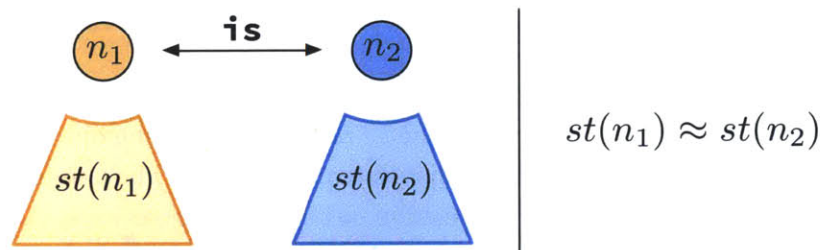


Figure 4.4.: Two nodes with an `is` relation are considered containers of equivalent atomic subtrees.

The relation $is(n_1, n_2)$, shown in Figure 4.4, states that n_1 and n_2 are containers of atomic data with equivalent type and role. In other words, the subtrees of n_1 and n_2 can be swapped, or one can replace the other. This is the analog of a symmetric print statement.

For example, an HTML theme may have an `h1` element representing the page title. A JSON object may contain key whose value represents a particular page title on your website. An `is` relation between those two nodes indicates that it would be semantically coherent to replace the contents of the `h1` element on the web page with the value of the object key in JSON. In other words, printing the JSON value at that particular place in the HTML tree.

We can imagine running that operation in the other direction, of course, or using that operation to respond to events on an already-rendered web page, resulting in a number of different uses for `is`:

4. Cascading Tree Sheets

Templating If t_1 (the tree containing n_1) was an HTML template, t_2 might be a JSON object containing dynamically fetched data to fill that template. Replacing the subtree of n_1 with the subtree of n_2 is equivalent to a print statement whose output fills the n_1 container.

Scraping If t_1 is a web page containing data to be extracted and t_2 is a data container assuming the shape of the data to be extracted, then replacing the subtree of n_2 with the subtree of n_1 is equivalent to "scraping" exactly one field from the web page into the receiving data element.

Reactive Programming If t_1 and t_2 are two data structures being synchronized in real time, then modifications within the subtree of n_1 or n_2 could trigger a change notification event which propagates across the *is* relation to ensure the other tree is kept in sync.

It may be helpful to further explain the *is* relation by pointing out how it differs in meaning from some similar relational statements that may be familiar to the semantic web community. The OWL relation `owl:sameAs`, for instance, states that n_1 and n_2 are the *same entity* [39]. The *is* relation, rather, makes a comment about the type and semantic compatibility of subtrees of the two nodes.

The `owl:equivalentProperty` relation states that n_1 and n_2 may be considered equivalent unary relations themselves. If we were to adapt this notion to the CTS data model, this would be making the statement that n_1 and n_2 could be considered equivalent unary relations, their subtrees as the arguments of these relations. The CTS *is* does not imply n_1 and n_2 have equivalent semantic roles however. On a commerce website, n_1 might contain information about a "special product" while n_2 contains information about a "discount product". Perhaps all product descriptions on the website are structured identically (subtree compatibility) but the roles of special products and discount products are incredibly different.

4.4.3. Graft — Structured Content Equivalence

The relation `graft(n_1, n_2)` states that n_1 and n_2 are containers of data with equivalent type and role, but that this equivalence is structured—further relations exist between nodes within the two subtrees. In other words, you could replace the subtree of one with the other, but that operation comes with strings attached: there

are more nodes in that subtree participating in relationships, too. Semantically, we will see that this is the analog of a symmetric `include` statement, or a parameterized template invocation.

For an example, consider the statement `print(x)`, which simply takes the value within `x` and places it somewhere. In many cases, we would instead like to encapsulate the printing of `x` within some larger piece of structure. A method call such as `showAlert(title, message)` might be provided by a UI toolkit to create a modal popup window and then fill the `title` and `message` slots in it with the provided data. Or a blog theme might encapsulate and store the HTML required to organize a blog comment inside a `comment.template` file, to be parameterized by the commenter name, email, and opinion.

All of these examples can be thought of as structured versions of `print`, or—in our new relational sense—the `is` relation. Displaying an alert box, or a blog comment, is essentially a request to `print` some stored piece of data, but to punch out slots in that data with further parameterization. `graft` represents this relationship.

A `graft` relation between the node wrapping the data (`title, message`) and the node wrapping the alert dialog box template indicates that one subtree may be swapped by the other, but further parameterization is defined below, in the form of more relations. In the alert box case, this would be an `is` relation between the `title` and the slot in the template where the title should go, and another `is` relation between the `message` and the slot in the template where the message should go.

4.4.4. Are



Figure 4.5.: Two nodes with an `are` relation are considered containers of equivalent lists.

The relation `are(n_1, n_2)`, shown in Figure 4.5, states that n_1 and n_2 are containers of equivalent lists, and each child of n_1 or n_2 are list items. In other words, it would be semantically coherent to treat the i^{th} child of one as sharing a special scope with the i^{th} child of the other, as they both represent the same item.

4. Cascading Tree Sheets

What *to do* about this relationship between n_1 and n_2 could be interpreted in a number of different ways. CTS uses an interpretation that allows `are` to be the symmetric relational equivalent of the `foreach` statement: one node represents the container of the list to be enumerated over and the other node represents the output form of the list enumeration. Specifically, the `are` relation does not imply that the subtrees should be swappable outright like `is` or `graft`, but rather that:

- The cardinalities of these two lists can be aligned. If $|n_1| \neq |n_2|$, then one list may either be trimmed or the other inflated such that they align.
- The relations within the subtree of the i^{th} child of n_1 and the subtree of the j^{th} child of n_2 should be ignored if $i \neq j$
- *As with the `graft` relation, the utility of `are` is revealed when it is composed with other relations in the subtrees beneath n_1 and n_2 .*

This is a rather abstract way to talk about a `foreach` loop, so let's look at an example. Say we have a simple design template of a bulleted list:

```
<div id="#template">
  <ul>
    <li>A list item</li>
  </ul>
</div>
```

And we also have a segment of HTML that contains some data:

```
<div id="#data">
  <div>Data item 1</div>
  <div>Data item 2</div>
  <div>Data item 3</div>
</div>
```

We can use `are`, `is`, and `graft` together describe how these two structures relate:

```
#data          :graft  #template ;
#data          :are    #template ul ;
#data > div    :is     #template ul li ;
```


The `graft` statement says that the `#data` and `#template` nodes can be swapped, but the swap should be a combination (based on further relations) of the two trees rather than a simple replacement. The `are` statement says that the `#data` node and the `ul` node are both equivalent lists. The `is` statement says that the `#data > div` nodes and the `li` nodes are containers of equivalent items.

The presence of the `are` statement provides instructions about how to interpret the $n \times m$ `is` relations instantiated by the `is` statement in the tree sheet. Relations between the subtrees of the i^{th} and j^{th} items of the `are`-related nodes are ignored if $i \neq j$. And by calling on cardinality alignment and modular child indexing, we can now use these three relations in a number of ways:

Templating. If we use the above example to merge the data HTML into the template HTML, then we have just performed the equivalent of invoking, with parameters, a template that contains a for loop and a print statement.

Scraping. If we use the above example to merge the placeholder data from the template HTML into the data HTML, then we have just performed the equivalent of scraping data out of a design and into a data structure.

Reactive Programming. Once we have used the above example to template an HTML page, we can continue reusing the relations to react to modifications to the trees. For example, the source HTML might change, suggesting a change in the output HTML is appropriate. Or the web user might alter the output HTML in the browser, suggesting that a change in the data HTML might be appropriate.

Consider the following scenarios:

- Deletion is the most obvious case. If the i^{th} child of either the data HTML or the output HTML is deleted, then so too should the i^{th} child of the other.
- Insertion of a child into the data HTML or the output HTML should cause the other to re-update its cardinality (by cloning an item) to reflect this new insertion. Any relations between these two new subtrees would then be processed.
- Modification of the contents of subtrees would be ignored by the `are` relation because this relation merely causes the related nodes to behave as

4. Cascading Tree Sheets

synchronized sets but does not specify (as `downtree` is relations might) particulars about how items in these sets are related.

4.4.5. If-Exist and If-NExist — Existential Dependency

Finally, we introduce two relations, `if-exist` and `if-nexist`, to act as the relational analog of an existential if and its paired else.

The relation `if-exist(n_1, n_2)` states an XNOR relationship between the truey -ness of the subtrees of n_1 and n_2 , and the `if-nexist` relation states an XOR relationship between their truey -ness. Table 4.4.5 shows a boolean truth table for these relationships.

<i>subtree(n_1)</i>	<i>subtree(n_2)</i>	if-exist (XNOR)	if-nexist (XOR)
falsey	falsey	true	false
falsey	truey	false	false
truey	truey	false	true
truey	truey	true	false

Truey-ness and falsey -ness are concepts left defined by each tree type. When a `if-exist` or `if-nexist` relation exists and the CTS engine wishes to manipulate trees as a result of it, it asks the nodes that participate to report their truth value, which they interpret.

As some examples of how various elements in the CTS implementation defines truey -ness and falsey -ness:

- **HTML checkbox:** the checked state is true, all others are false
- **HTML input element:** a non-empty value is true, all others are false
- **HTML element:** No children or a CSS visibility of none is false, all others are true.
- **JSON Dictionary Attribute:** a value that parses as a non-false value in Javascript is true, all others are false
- **JSON Array:** an array of zero length is false, else true

4.5. Example Scenarios

From this list, it is clear that what is going on is a tree-based form of operator overloading. The object types on which the overloading occurs are different kinds of tree nodes that may occur in the CTS universe, and operators being overloaded are XOR and XNOR.

For example, perhaps a particular data object has a boolean `admin` flag, and the web design has a special sidebar to show for administrators. The CTS author could place a `if-exist` relation between the `admin` variable and the sidebar.

There is a final twist, made necessary by the fact that CTS statements are not relations themselves but rather contain queries that produce relations. If a selector's query against a tree results in the empty set on one side, but a non-empty set on the other side, the CTS engine creates a special set of relations bound to a singleton `NonExistantNode`. An analogy might be drawn to the `None` object in Scala or Haskell. This enables the `if-exist` and `if-nexist` relations to toggle the presence of design elements in a page not just based on the value of present data, but on the existence or non-existence of data. Since a major (and unique, among template languages) use case of CTS is *post hoc* retargeting of pages, and since such retargeting often requires handling of schema inconsistencies between pages, the ability to modulate the presence of design elements based on the existence of data elements is extremely useful.

4.5. Example Scenarios

Here are several examples of how we might draw relations between different data structures and what we might hope to do with them. Section 4.6 will then provide an overview of the CTS execution model, and Appendix B formalizes it.

4.5.1. A Basic `is` Relation: “This this is like that thing.”

As a basic example, let's look at several ways an `is` relation can be applied. Say we have a four documents, a spreadsheet containing information about a company, a JSON file containing information about another company, and two web page mock-ups for sale to companies looking for a web site. Figure 4.6 shows subsets of the hypothetical trees representing these documents and `is` relations between them.

In Figure 4.6, the `is` relation does not represent that the *value* of the related nodes is the same value. In fact they are quite different values: the two web page mockups

4. Cascading Tree Sheets

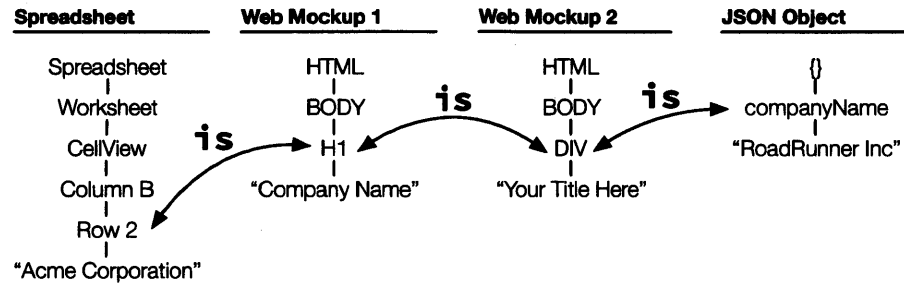


Figure 4.6.: Four hypothetical documents all containing information about a company. Here, we see how the *is* relation might be used between nodes whose subtree is the company name.

contain fake company names, and the spreadsheet and JSON file each represent different companies.

Instead, the *is* relation represents that the subtrees of the related nodes are structurally equivalent. Cell B2 in the spreadsheet contains a subtree which could be sensically swapped the H1 element of Web Mockup 1. Such a swap would damage the *data instance* represented by Web Mockup 1 (which is a dummy company) but it would retain the semantic coherence of the tree.

We could turn this web mockup into the actual web page for the company by performing swap. The resulting HTML tree would be HTML → BODY → H1 → Acme Corporation. If this swap were done dynamically in the browser, then the company spreadsheet would serve as a dynamic data source for the web page. The same could be done with the JSON file and Web Mockup 2.

4.5.2. A graft with a nested *is* : "Including a template to wrap my data."

Let's zoom on a more complex view of Web Mockup 1 and Web Mockup 2. Perhaps they both express a fancy title for the two respective companies. Figure 4.7 shows how a *graft* relation can be nested with an *is* relation to navigate these structural differences. The *graft* relation demarcates the containers for the company title and all its surrounding design. The *downtree is* relation demarcates the container for the actual string representing the company title.

Considered together, we might use these two relations to rewrite the two trees

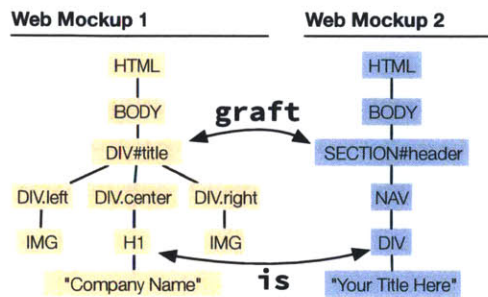


Figure 4.7.: A more complex view of Web Mockup 1 and Web Mockup 2 in which nested is relations exist.

in four different ways, shown in Figure 4.8. Perhaps we merely swap the subtrees of **graft**, resulting the first two combinations. If this were done, each company's web page would receive both the style and the company name of the other. The second two rewritings also swap the dntree **is** relation, causing the style to be transferred between the two trees but the company name to remain the same.

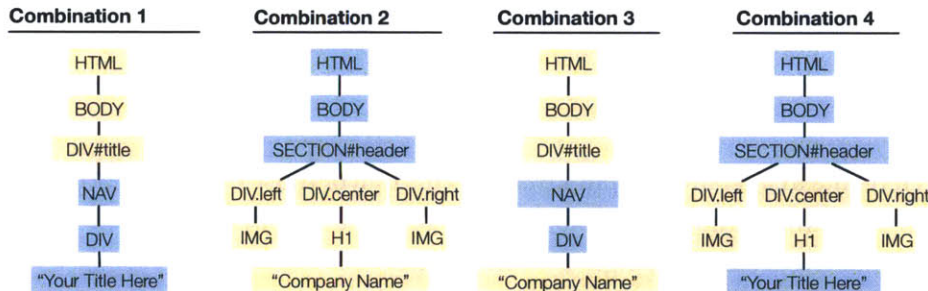


Figure 4.8.: Possible rewritings of the forest in Figure 4.7. In all four cases, an odd number of swaps has been performed based on the **graft** subtree. In 1 and 2, an even number of swaps has been performed on the bottom, and in 3 and 4, an odd number of swaps has been performed on the bottom. Semantically, cases 1 and 2 correspond to "opaque transclusion" and cases 3 and 4 correspond to template invocation.

4. Cascading Tree Sheets

4.5.3. A list of Todo items

Now we'll see how an are relation might apply using a Todo list, in which a bulleted list of items is related to a JSON data structure containing Todo items. Figure 4.9 contains the HTML and JSON trees which might represent a Todo list. The are relation is drawn between the ul element on the HTML side, which serves as the container for the items in the UI, and the array element in the JSON structure, which is the container for JSON dictionaries which contain individual tasks.

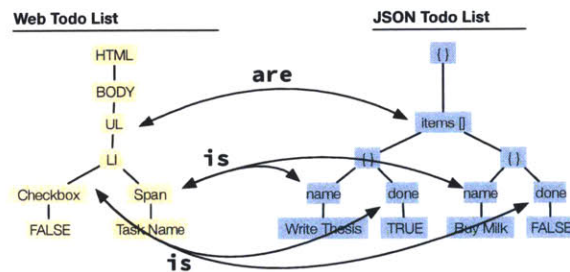


Figure 4.9.: An example combination of an are relation and is relations. The are relation represents set equivalence between the ul node in HTML and the items [] object in the JSON. The completion state and task name slots in the subtrees are also related. The CTS Engine takes care of making sure that the i^{th} items of each collection share scope with each other.

Here, the are relation implies that the children of those containers are equivalent lists of children from a type standpoint: both are lists of Todo items. Each direct child node represents a single item in the list.

The is relations within the subtrees of two are-related nodes take the same meaning as before. The checkbox in the HTML Tree of Figure 4.9, for example, has type equivalence with two separate nodes in the JSON Tree, as does the span element.

The are relation and downtree is relations, when considered together, give us the ability to transfer one list of items into the structure of another. Figure 4.10 shows two possibilities of this. At left, we see data from the JSON object transferred into the structure provided by the HTML tree. At right, we see the data from the HTML tree transferred into the structure provided by the JSON object.

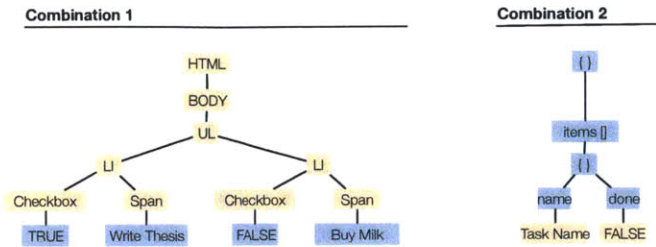


Figure 4.10.: Two different ways that the CTS Engine might render Figure 4.9 depending on which tree was the focus. I.e., which tree represents the “foreach” statement and which tree represents the data.

4.5.4. Hiding Completed Items

Suppose we wanted to designate that a “Done!” label should be placed next to Todo items only if they are complete. There would be two possible strategies for expressing this relationship using the previous Todo list example, both shown in Figure 4.11. Both involve first adding the structure `Done!` to the HTML tree inside the `` element.

The first option is to relate the `span` to the `checkbox`¹. The second option would be to relate the `span` to both `done` nodes in the JSON data structure. Either strategy states that the subtrees of the related nodes should have the same `falsey`-ness or `truey`-ness. If one is `falsey`, the other should be `falsey`, and if one is `truey`, the other should be `truey`.

The only difference is that the first option keeps the relations within the same tree—the equivalent of predicating a UI element’s state on the state of another UI element rather than the state of the model. Either practice has its advantages. Keeping the relation within the tree keeps this relationship independent of whatever data layer may be connected. Drawing the relation across the trees makes it clear that one element is the model and one is the view.

Recall that each CTS node has the liberty to interpret its own notion of `truey` and `falsey`. In Figure 4.11, the `checkbox` node interprets this as its checked state, the

¹There is no `checkbox` element in HTML of course. In these figures we use `checkbox` as a concise way to represent the element `<input type=“checkbox”>`, which would be expressed in the CTS tree as a two-level subtree. An unchecked box, for example, would be the subtree `input:Node(type:Attribute(checkbox:String), value:Attribute(FALSE:Boolean))`

4. Cascading Tree Sheets

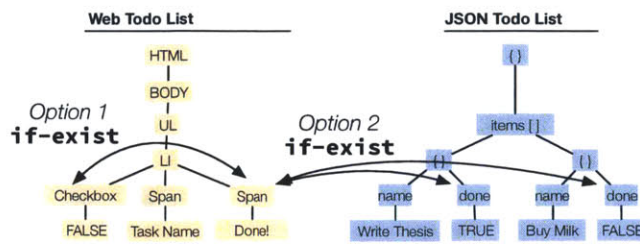


Figure 4.11.: Two different ways conditional visibility could be expressed using relations, given that the are and is relations from Figure 4.9 are also in existence to synchronize state between the trees.

spaninterprets it as its CSS visibility state, and the attribute node interprets it as the presence and truey -ness of its child.

4.5.5. Composing Multiple Trees

Finally, Figure 4.12 depicts a larger example of composition between three trees. This example mirrors the kind of composition that occurs when piecing together a web page today. One trees represent the containing page, another trees represent dynamic data, and still another trees represents a "sub-template" that encapsulates a the design of a particular region of the display that maybe reused many times across different pages.

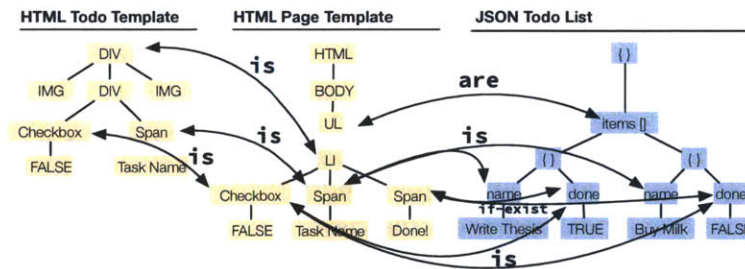


Figure 4.12.: CTS can be used to compose multiple trees, some acting as themes that wrap around content, others acting as data that fill the themes, and yet more acting as widgets or sub-templates that are inserted into a theme, perhaps with parameterization.

This figure is complex when drawn as an actual tree. The CTS syntax will make expression of these relationships more concise using CSS selectors, however.

4.6. Rendering Web Pages

Let's finally return back to the notion of web templating. Here we will make a few high-level observations that are formalized in detail in the Appendices for readers who are interested.

At the highest level, the CTS Engine takes as input some *root* tree, which is the HTML tree you loaded in your browser. This *root* tree includes CTS statements in various ways: by linking to tree sheets, by inlining them, etc. In turn, these CTS statements name and reference other trees out on the web.

The first job of the CTS Engine is to construct a forest from all this information. It fetches any referenced trees and applies CTS statements against those trees to create relations. Once all required data has been loaded and all relations have been created, it begins the rendering process.

The rendering process can be thought of as a black box function (full formalized details are provided in Appendix B) that takes the *root* tree, all the other trees, and all the relations, and produces a new tree to be added to the forest.

render(root, trees, forest) : newTree

Section 4.4 on semantics showed that the CTS relations were designed to be symmetric relational analogs of templating commands. The examples that followed shows ways in which a web author might apply those commands to data in the same way a web author might create templates.

Here, we describe a very important property of the CTS rendering process: that it is lossless with respect to the *root* node and its relations (We formalize and prove this notion in Appendix D). Lossless rendering in the context of CTS means two things:

- The output tree of the rendering process can be used just like the *root*, to the same end result, even if the other trees are different next time around. In other words, the core template (conditioned on the *root* as entry point) is completely preserved end-to-end. Contrast this with template rendering as practiced today, which removes much or all of the template information, requiring exten-

4. *Cascading Tree Sheets*

sive reverse engineering for reuse. When rendering under CTS, a visitor to the page who only sees the final output has a reusable artifact as good as the original input.

- The output tree of the rendering process contains clones of all the relations that caused it to appear the way it does. Put another way, the output tree contains enough information to provide the provenance of all of its nodes. This is the basis for using the relational layer CTS provides to power reactive read-write interfaces (Chapter 6). When the source trees change, they can trace across relations to modify the output tree. And when the output tree changes, it can trace backward across those relations to update the sources.

This represents the important parts of the rendering process, at least for the goal of quickly moving the discussion to the interesting usability ramifications. If it was too brief for your tastes, but you are willing to work with partial information, simply keep in mind that we've constructed a symmetric relational templating language, and you will be able to follow along. If you would rather see all the particulars first, please refer to the Appendices.

5. Static Content Authoring

Chapter 4 laid the groundwork of a relational language for web content called Cascading Tree Sheets. CTS is authored like CSS and makes relational statements about how structures on the web relate to other structures on the web. These annotations are symmetric and declarative, but we saw in Chapter 4 how a rendering engine can process them to produce web designs from component material, just a web template engine.

This chapter runs with that basic observation---that CTS can be used as a web template language---and investigates the application of CTS to static content authoring. In other words: *“another template engine, so what?”* This chapter will show that the relational model CTS relies on enables a form of web templating with very different characteristics than the existing body of practice. It will demonstrate (with user studies, static analysis, and reasoned examples) that CTS' relational approach provides improved authoring, reuse, and maintenance of static web content and style.

This thesis uses the term *static authoring* to refer to content and style that is static from the perspective of the author, not the software system serving the page. A blog post and blog theme may be stitched together on-demand by a Content Management System, but the spirit of these two files is write-only content from the perspective of the author, and read-only content from the perspective of the reader. The output document is static. It only computation at run-time because the content has been stored in a way to separate concerns between the content and design components.

This definition of static authoring, broader than the term is ordinarily used, thus encompasses most of the creations that bring us to the web as readers rather than participants: blogs, articles, personal pages, wiki pages, and how-to articles are forms of web content that are static using our definition, even if we often use dynamic gears to serve them to the reader.

A useful lens with which to view static authoring is the tension between content

5. *Static Content Authoring*

and presentation. Static HTML documents necessarily contain both. Even if an author completely forgoes efforts to add visual styles to a document, there are still presentation issues concerning document structure: paragraphs, articles, titles, and so on. And designers who are completely disengaged from content creation still must populate their presentations with mock data, even if that data consists entirely of stock images and *lorem ipsum* copy.

One way to organize an exploration of the usability of static content authoring is thus to divide authoring activity into the two separate roles of content authoring and presentation authoring. How easy is each role to create? To what extent can they be reused? And how well encapsulated is this reuse to change?

This chapter uses this framing of the task to investigate the CTS approach to static authoring from three perspectives:

- **Authoring and Efficiency**, in which we demonstrate that CTS' relational approach improves the speed of authoring both content and design by enabling a complete separation of concerns between the two. We also make the case that the particular mechanism CTS enables for this separation of concerns, which we dub **mockup-driven development** (MDD), creates a better user experience than traditional templating.
- **Reuse and Portability**, in which we demonstrate that the web-relational enables CMS-style reuse of themes and content without the system, freeing authors from the need to rely on large client-server application frameworks with vendor lock-in. We investigate this facet from angles. First, we show how the mockup driven development approach can be applied to the creation and publication of web widgets. Next, we demonstrated how theming communities can be created around any set of pages with a similar domain even if they were not created with CTS and MDD in mind.
- **Maintenance and Independence**, in which we show how the practice of using CTS relations as the binding layer between web widgets (or themes) and the authors who wish to invoke them provides a layer of encapsulation that is missing on the web today. Specifically, it enables the HTML structure that a page depends on to be linked by reference, just like CSS and Javascript. An example from the Twitter Bootstrap library is used to show the benefits of this additional capability in practice.

5.1. Authoring Efficiency and Mockup Driven Development

This section shows that authoring HTML with CTS is more efficient than authoring ordinary HTML without it. Using CTS, an HTML page can be split into two separate HTML documents, one to contain a presentation mockup¹ of the final document and another to contain the raw content. Relations stored in a separate tree sheet specify how regions in the content document relate to the presentation mockup. A CTS engine on either a web server or in the client merges the two together for display to the user.

This approach is in contrast to the way we write ordinary HTML today, which conflates content and presentation. Any modern web page necessarily contains some HTML that exists solely to demarcate content and other HTML that exists solely to scaffold presentation elements. Intuitively, these two roles of HTML are at odds with each other from the perspective of efficiency. When authoring content, most (though perhaps not all) HTML that is solely devoted to presentation is effectively noise the author has to wade through, and vice versa. An efficient authoring method would instead partition these two concepts by encapsulation, so presentation authors can focus better on presentation and content authors can focus better on content.

We will first quantify the impact of conflating content and presentation by adapting the notion of signal-to-noise ratio for web authoring. Using data scraped from WordPress sites, we will show that the HTML is getting harder to edit as time passes. Next, we discuss CTS' relational approach to presentation and content encapsulation, which we call **mockup driven development**. Mockup driven development is similar to web templating, but enables new ways of authoring and publishing content that are not possible with traditional templating. Since template languages are also mechanisms for partitioning content from design, the novelty in this chapter is the manner in which CTS performs the separation and what can be done with it. Finally, we show the results of a user study empirically supporting the intuition that separating presentation from content with mockup driven development results in more efficient HTML authoring.

¹Note: this thesis often uses the words presentation and design interchangeably.

5. Static Content Authoring

5.1.1. Thinking about HTML Authoring as Signal-to-Noise

In signal processing "signal-to-noise ratio" (SNR) refers to the power of a signal compared to the background noise. We can adapt this concept for static editing by thinking about it instead as the percentage of an HTML document relevant to one's authoring role compared to the percentage of the document that is irrelevant. For example, when editing content in a document, ideally we would not have to navigate through a large amount of presentation scaffolding. And when editing a web design, ideally we would not have to navigate through lots of content.

For any HTML document, let's define $nodes(content)$ as a function which returns the number of HTML nodes dedicated to delineating content fields, and $nodes(design)$ as a function which returns the number of nodes dedicated to scaffolding the page design but are not necessary for delineating content fields. For the tasks of content authoring and design authoring, we can then adapt the concept of SNR as the number of nodes dedicated to that role divided by the total number of nodes. For design editing, the SNR is $\frac{nodes(content)}{nodes(content) + nodes(design)}$, and for content editing the SNR is $\frac{nodes(design)}{nodes(content) + nodes(design)}$. While different than the classic SNR calculation, this formulation is useful in that it will always be between 0 and 1, with 1 representing a document unsullied by the other perspective.

We can reason out that amount of HTML scaffolding to describe a data structure, or in HTML ($nodes(content)$) tracks the shape and size of that data structure linearly. For example, a single HTML node must be dedicated to every object, every array, and every array property to enable a complete preservation of a JSON structure, as seen in Figure 5.1. Any further HTML nodes past that point are superfluous. A blog post containing a title and article body would require at most three HTML elements: a node to represent the post object, a node to represent the title, and a node to represent the article body.

On the other hand, $nodes(design)$ may be arbitrarily large with respect to $nodes(content)$. As an extreme example page might have a single content node, perhaps a title, surrounded by thousands of nodes of design.

We can also show that presentation-centric HTML scaffolding will always be necessary no matter how many new CSS rules are added. Consider the simple thought experiment of a new CSS rule to add a circle next to an HTML node. Adding two circles next to an HTML node would require an extra HTML node to participate (to

5.1. Authoring Efficiency and Mockup Driven Development

JSON	HTML
[
{	
name: "Spock",	<h1>Spock</h1>
position: "X0"	<h2>First Officer</h2>
}	
,	
{	
name: "Kirk",	<h1>Kirk</h1>
position: "Captain"	<h2>Captain</h2>
}	
]	

Figure 5.1.: Expressing JSON data as HTML requires one element for every object, every array, and every object property. The result is what we call a *content page*—a raw data object like JSON but expressed as HTML, viewable in the browser, and editable by ordinary HTML editing tools.

provide the extra circle). No matter how many extra CSS rules we add for circles—a 2-circle rule, a 3-circle rule, an n -circle rule—we can always create a design with $n + 1$ circles requiring this extra node.

And finally we can show empirically that designs seem to be getting more complex over time. The schema of Wordpress, a popular blogging platform, has not significantly changed since its inception. It supports blog posts and web pages, each with a title, author, body, and tags. But if we plot the size in kilobytes of WordPress theme files, shown in Figure 5.2², we see that themes in 2014 are roughly three times larger than they were in 2008. Since the content schema of WordPress is fixed, only increases in presentation scaffolding can account for this change. This data was scraped for this thesis work, and binary files (images, flash documents, sounds) were removed before measuring sizes.

So the amount of HTML dedicated to content is firmly bound, while the amount dedicated to presentation appears to be increasing over time. This causes a problem for any author managing plain HTML files. The only way to shield content editing activities from the growing burden of design is to incorporate template languages, but that requires that the design HTML be rewritten in a non-HTML language and

²Prepared for this thesis

5. Static Content Authoring

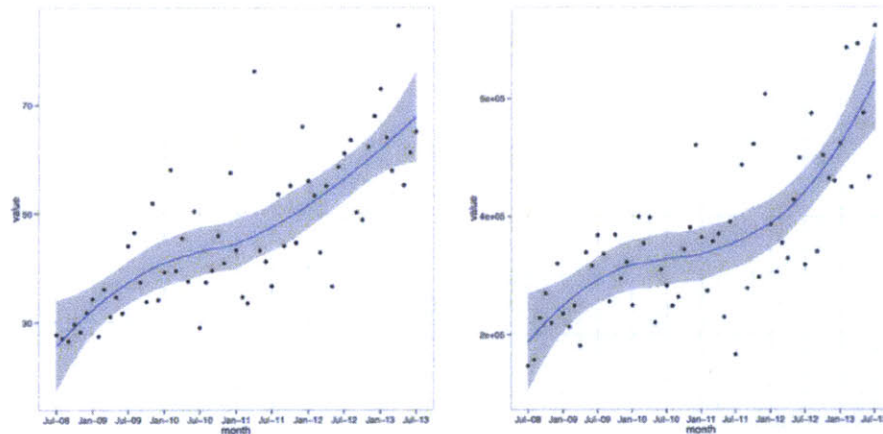


Figure 5.2.: Design scaffolding is growing larger over time, as evidenced by the growing size of WordPress themes (binary files such as images were removed for this measurement). Since the content schema of these themes is fixed, only design can account for the change.

the content HTML be re-written in a non-HTML data format (such as JSON)³. Authors who wish to maintain a web page in the traditional style, as a set of HTML files on an HTTP-accessible file system, are stuck conflating the two types of HTML in the same document.

We can see the impact of this by looking at the SNR of some of these WordPress pages when they are finally rendered to the web. We downloaded the top five featured blog templates on the Wordpress home page and extracted the HTML structure which wraps exactly one blog post. For each blog post template, we then tallied the total number of DOM nodes, the number of content nodes (the number of distinct information fields plus one node to represent the blog post object), and the number of presentation nodes (the difference between the two). Table 5.1 shows the result.

5.1.2. Mockup-Driven Development

The CTS approach to static content and presentation authoring is to split an HTML document into two separate, self-contained documents: a content document and

³While CTS does require that a separate tree sheet be authored, which is not HTML, the design and content documents remain ordinary HTML, which is not the case with traditional templating.

5.1. Authoring Efficiency and Mockup Driven Development

Theme	Nodes	Content Nodes	Design Nodes	Content SNR	Design SNR
Responsive	16	7	9	0.44	0.56
Pinboard	18	6	12	0.33	0.66
Buttercream	13	6	7	0.46	0.54
Twenty Eleven	23	7	16	0.3	0.7
Montezuma	22	10	12	0.45	0.55

Table 5.1.: Design and content signal-to-noise ratios of a blog post for the five top featured Wordpress themes.

a presentation mockup. We call this pattern of development *mockup-driven development* (MDD) because both the content and presentation artifacts more closely resemble creative mockups than the database rows and template files used in common templating practice today.

The *content document* is what the casual observer might describe as a “1993-style” web page: a simple, bare-bones HTML document containing just content. This document contains only the HTML necessary to encode the structure of the content as HTML: in JSON terms, one node for each object, array, and property. This means its signal-to-noise ratio is a perfect 1, compared to the SNRs below 0.5 in Table 5.1.

The *presentation mockup* contains all the design scaffolding and Javascript code necessary to achieve the desired presentation. It could either be an actual page, live on the web (a university department might maintain a standard academic home-page mockup for example), or a privately maintained mockup.

These two documents are then related to each other via CTS relations that mark which elements of content correspond to which mock elements in the presentation mockup. The title of a web page in content document, for example, is related via an *is* relation to the element in the presentation mockup containing the title. Each collection of content elements is related to the container of that collection in the presentation mockup with an *are* relation, and so on.

The result of this process is three files: two HTML pages and a tree sheet. To combine the three files into a web page for presentation, the author simply publishes the three files to the web and includes the tree sheet with a `link` element in the head of the content page just like a style sheet. A rendering engine in the browser or on the server then combines these two documents for display to the reader.

This general pattern can of course be applied to many content and mockup docu-

5. *Static Content Authoring*

ments participating to create the same page, not just two. One presentation mockup might contain the overall page layout, another might contain the design of a blog post, and yet a third might contain the mockup for a particular widget that appears on the page. One content document might contain the header and footer copy to be reused across a whole site, while another might contain just the copy for a specific article.

On the surface, this sounds just like web templating. And it is! But this relational approach to templating results in a few important differences to templating today.

Web templating today typically employ a three-stage process. In the first stage, designers use a tool like Adobe Photoshop to create design proposals. In the second stage, a full HTML mockup is created for the selected design. And in the third stage this mockup is cut into small fragments and rewritten in a template language like PHP.

This process today is an unfortunate consequence of the using the right (existing) tool for the job: designers simply find Photoshop a more efficient way to iterate through visual ideas. HTML is the target language for implementing those ideas, but program-like template fragments are the required input to the current technology we have to encapsulate and reuse presentation elements. By the time a design has reached the template fragments stage, its parts can no longer be sensically rendered in the web browser in the absence of a running system with production data, and the design asset now depends upon the schema of that data. This also creates a barrier between design tasks and development tasks. Changes made to the design need to be manually trickled through the pipeline to the templates. When this involves multiple people, the challenges are compounded.

Mockup Driven Development with CTS fixes many of these problems. First, MDD enables an approach closer to HTML-everywhere than the design and deployment process today. The HTML mockup created in stage two of the process described above becomes the final production artifact without any further changes. Because it is ordinary HTML, the mockup renders in a browser in the absence of a running system. But since the running system uses relations to reference and bind to this document, changes to the mockup result in changes to the running system as well. This eliminates labor and places designers closer to the production artifact.

The same improvement is found for content authors. Data serializations like JSON are not optimized for viewing or editing by hand. JSON, for example, does

5.1. *Authoring Efficiency and Mockup Driven Development*

not support multi-line strings. By enabling authors to store raw data as HTML and stitch it at run-time to designs, authors can view, edit, and publish their data in a format designed for human use. It may be that limited amounts of HTML to serve as markers (e.g., an `<h2>` element and there) inside a document would lead to easier editing than a minimalistic JSON-style content document. In this case, the HTML content document would serve both to demarcate data structure and provide a lightweight UI to signpost the editing process. Multiple documents, not just a single content and presentation document, can be merged and composed as well. A simple blog content page might wrap itself in one theme, and then wrap each blog post in another theme. One of those posts might further include some other custom design.

A second major distinction to web templating today is that MDD externalizes the template instructions (in this case, relations) and stores them in a reusable tree sheet. This further enables the content and presentation documents to be singular in their focus. It also adds a layer of indirection that makes switching presentational styles a simple matter of linking to a different sheet.

Finally, the particular MDD approach using CTS relations provides a viable model of bringing the kinds of design and content separation that CMSs and programmers enjoy to the web at large. The approach mirrors that taken by CSS: it is visible to the browser (instead of in a black box on the server), reusable between sites, and declarative. It builds upon the existing authoring practice of HTML in a strictly additive fashion. And it builds upon the existing corpus of HTML documents without requiring a format change. This final property has an interesting consequence for reuse: any HTML page already published to the web becomes a candidate for incorporation and reuse as a presentation mockup for someone else's content. We explore this possibility in Section 5.2.4.

5.1.3. **Content Authoring User Study**

We evaluate this reduction in complexity with a user-study that compares the MDD approach to ordinary source editing. We found three university course web pages that appeared, based on inspecting their HTML, to be written and maintained by hand. These three are shown in Figure 5.3. We then downloaded the HTML page and created two variants:

The HTML Method was an exact duplicate of the original. This is the document

5. Static Content Authoring

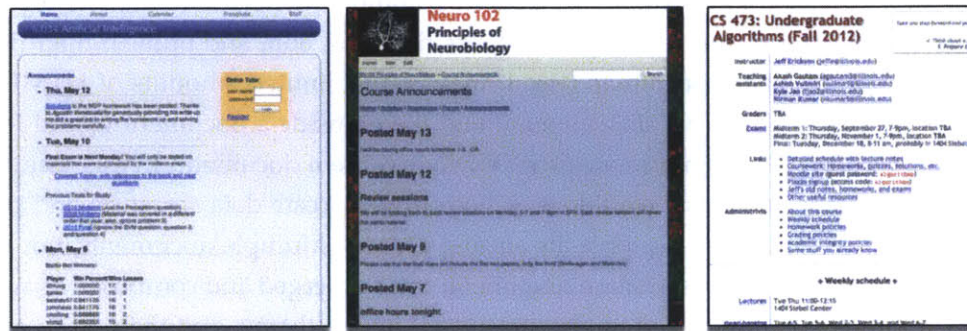


Figure 5.3.: Screen shots of the three course pages used in our study.

that professors or TAs would have to edit when adding a new course announcement normally.

The HTML-Simple Method was a hand-created copy of each course page in which just the announcements were extracted and the HTML structure minimized to just four HTML elements per announcement (a wrapper, title, time, and body).

An example of the simplified HTML document is:

```
<html>
<body>
  <h1>Announcements</h1>
  <section>
    <article>
      <time>Thu, May 12</time>
      <div>
        <p><a href="handouts/spring11/homework4-sol.pdf">Solutions</a> to
        the MDP homework has been posted. Thanks to <em>Agustin
        Venezuela</em> for generously providing his write-up. He did a
        great job in writing the homework up and solving the
        problems carefully.</p>
      </div>
    </article>
  </section>
</body>
</html>
```

5.1. Authoring Efficiency and Mockup Driven Development

```
<article>
  <time>Tue, May 10</time>
  <div>
    <p><b>Final Exam is Next Monday!</b> You will only be tested
    on materials that were not covered by the midterm exam.
    <p><center><a href="final.txt">Covered Topics, with references
    to the book and past questions</a></center></p>
    <p>Previous Tests for Study</p>
```

... (and so on) ...

For the HTML-Simple method, participants only saw this simpler HTML page, but as an exercise, we authored tree sheets to connect these content documents back to the design mockups to verify that this separation of concerns could be re-joined by CTS. Here is the tree sheet for the example above, written to be rendered from the context of the design page⁴.

```
@html simple-page http://localhost:8000/simple-page.html;
@html full-page http://localhost:8000/full-page.html;
```

```
simple-page | section
:are
full-page | div.announcements ;
```

```
simple-page | article > time
:is
full-page | div.announcements p.schedhead ;
```

```
simple-page | article > div
:is
full-page | div.announcements p:nth-child(2) ;
```

⁴Note the lack of a *graft* relation. Had we wanted to combine the two documents from the context of the content page we would have required more relations to wrap the simple HTML and then wrap each enumerated item.

5. Static Content Authoring

Participants that we recruited were asked to copy, paste, and edit course announcements on a university class page. We chose the copy, paste, and edit operations because they are the quintessential primitive tasks of anyone maintaining source-level web content, and we chose university class pages because they are a typical (and familiar) domain in which by-hand HTML editing frequently occurs.

Our study consisted of 17 computer professionals (both graduate students and industry professionals) who program for a living, all of whom reported having had experience authoring in HTML before. For each action (copy, paste, edit), for each method (HTML, HTML-Simple), subjects performed a practice task and then two timed tasks, for a total of 12 data points per user ($3 \times 2 \times 2$). Subjects were randomly grouped into two blocks, which differed in the order in which methods were presented to the subject. Block one performed tasks using traditional HTML editing first (which we'll call the **HTML** method) and block two performed tasks using the simplified content document first (which we'll call the **HTML-Simple** method).

The screenshot shows a web browser window titled "Web Template Editing Experiment". The main content area is titled "Copy a Course Announcement". Below the title, there is a paragraph: "When you begin, you will see the HTML for a university course page at right." To the right of this text is a code editor showing HTML code for an announcement. The code includes a date "Thu, May 12", a link to "Solutions" for homework 4, and a paragraph of text: "Thanks to Agustin Venezuela for generously providing his write-up. He did a great job in writing the homework up and solving the problems carefully." Below this is another announcement for a "Final Exam" on "May 10", with a link to "Covered Topics, with references to the book and past questions" and a link to "Previous Tests for Study".

On the left side of the interface, there are two numbered instructions:

1. Select the **first course announcement** and copy it to the clipboard.
2. Paste it into the text box at the bottom.

Below the instructions is a note: "Note: Your task is to copy the HTML for the whole announcement, not just the text." At the bottom of the interface, there is a "Paste Here:" label and a text box containing the HTML code for the first announcement, which is a subset of the code shown in the editor above. A "Next Step" button is visible in the bottom right corner.

Figure 5.4.: Copy task using the **HTML-Simple** method. The only difference to the **HTML** method is the HTML the subject must navigate.

5.1. Authoring Efficiency and Mockup Driven Development

	HTML	HTML-Simple
Copy	53	13.6
Paste	31.1	8.5
Edit	25.6	12.6

Table 5.2.: Mean time to completion, in seconds, for each method and action type.

The experiment was automated as a web application which guided participants through each step. For each discrete task, the subject was presented with a briefing of the steps they would perform, followed by the HTML source for that task, which was syntax-highlighted and line-wrapped to improve readability. For all tasks, we recorded the time between the user clicking a "Begin Task" button (after reading the briefing) and clicking a "Finished" button. Task descriptions were:

Copy Task. Each copy task asked a user to copy either the first, second, or third course announcement from the HTML source and paste it into a text box. Example shown in Figure 5.4.

Paste Task. Before starting the paste task, users were given an HTML fragment to copy to the clipboard. They were then asked to paste that fragment after the first, second, or third course announcement in the HTML source that was revealed.

Edit Task. Users were asked to change the title of an announcement. To do this, they had to find the announcement in the HTML source and then edit it.

We find that for all tasks, the simplified editing approach results in faster edits, as depicted in Figure 5.5. A two-way within subjects analysis of variance found this interaction between the method and the completion time to be significant: $F(1, 2) = 23.89$, $P < 0.001$. Table 5.2 records mean time to completion for each method and action.

These results suggest that communities that edit HTML would benefit from dividing their HTML authoring workflow into separate design- and content-centric documents.

5. Static Content Authoring

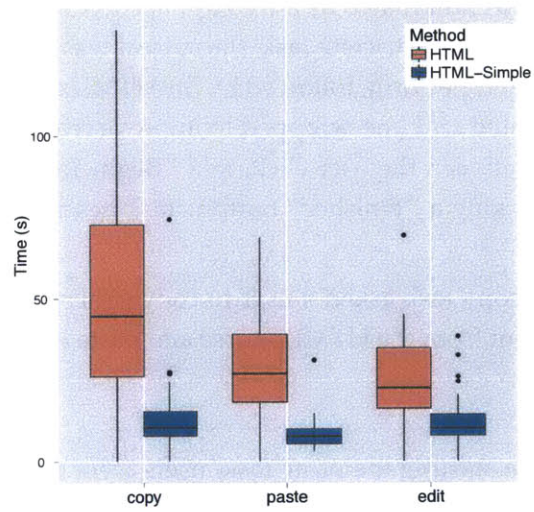


Figure 5.5.: Whisker plot results for the two methods on copy, paste, and edit actions. Measurements are in seconds and dots represent outliers.

5.2. Reusing Content and Presentation

While this study only tested the positive impact of MDD on content editing, we assume that (by symmetry) there is a similar benefit to presentation editing.

5.1.4. Summary

A modern web page contains intermingled content HTML and presentation HTML. Separating these two concerns yields benefits to authoring usability, but current practice for performing this separation requires departing from ordinary HTML authoring and adopting programming languages and data serialization formats. Authors that prefer to use HTML are left sorting through web pages with low signal-to-noise ratio from the perspective of both content and presentation.

This section showed that CTS relations can be used to perform the same kind of content-presentation separation as template languages but with the added benefit of (1) externalizing the template instructions into a separate tree sheet, and (2) relying on ordinary HTML for both the content and presentation documents. This enables:

- Existing WYSIWYG editors to manipulate them
- Browser to view them, as either simple content or design demos.
- Others to link and reuse content and design from other pages.

5.2. Reusing Content and Presentation

This section shows that web pages authored with CTS enables an interesting set of reuse possibilities. We will develop the idea of a CTS microformat to mirror the existing practice of CSS microformats, and then show how these microformats can be used to share and reuse design. For widgets, CTS microformats enable HTML to be used as a calling convention to invoke rich page UI elements. For themes, CTS microformats enable theming communities to emerge organically, sharing each others designs based only on common application of CSS classes.

All of these practices are possible today, but they typically require incorporation of large application-layer frameworks (e.g., a content management system for themes), programming by the end user (e.g., for widgets), or both. The novelty of the CTS approach is the properties of the end-user reuse environment. Simple application of CSS classes to an HTML document can, in the presence of a tree sheet,

5. Static Content Authoring

associate that document with relations which perform complex structural transformations, importing new content and even Javascript. And themes can be created for arbitrary information domains without explicitly defining a data source and data schema.

5.2.1. CTS Microformats

In Section 5.1, we saw how an HTML document could be divided into separate content and presentation documents and then bound together by a tree sheet. This practice, called **mockup driven development**, enables encapsulation of content from presentation while enabling both to remain ordinary HTML. The selectors in the tree sheet used with MDD could be based on any document property, such as structure, node type, or semantic class. But some of these selector styles are clearly more amenable to reuse than others. The selector `p:nth-child(2)`, which selects paragraph elements which are the second child of their parent, is clearly less fit for reuse than the selector `.important`, which selects any element tagged with the CSS class `important`.

We might imagine setting out to create a tree sheet constructed entirely with CSS class-based selectors on the side of the content document, and we would carefully design them with the intent of forming a reusable set of classes. Instead of rules like

```
@html content this;
@html presentation http://www.example.org/widget-definition.html;
content | p:nth-child(2) :graft presentation | section:first-child > div;
```

We would write rules like:

```
@html content this;
@html widget http://www.example.org/widget-definition.html;
content | .important :graft widget | #importantWidget;
```

Seemingly little has changed in terms of implementation: we've just been more careful about providing a clean set of CSS class-based selectors instead of selectors with structural dependences on the page at large. But this change supports a new mode of mockup-driven development that is framed around communities. If communities of interest agree on common sets CSS classes to represent a certain concept,

like a business card or calendar item, then they can all benefit from trading designs and functionality surrounding those item types.

We will call this use of CTS a *CTS microformat* after the established practice of CSS microformats. CSS microformats are an idea from the semantic web community that aims to make annotation of data semantics in a web page more accessible [55]. A CSS microformat is a curation of CSS classes that, used together, implies that the content within those nodes is an instance of a particular data schema. For example, the vCard CSS microformat encodes contact information using classes like `url`, `given-name`, and `tel`. If someone uses those particular CSS classes on a web page, the implication is that they are communicating information about the semantics of the data within the nodes carrying those classes.

While CSS microformats are a convenient way to add data semantics, recall that CSS alone can not arbitrarily style the HTML structures which have been annotated. That requires the ability to rearrange, add, and delete HTML structure. CTS provides these additional abilities⁵, so that the same vCard microformat used to convey semantics can also be used as a calling convention into rich, visual widgets or themes, the invocation of which is powered by CTS.

Any web page that includes a tree sheet using a microformat can be said to have imported that particular widget or theme. In the example beginning this section, that calling convention into the widget is simply the CSS class `important`. Any page which imports this widget can invoke it by attaching that class to a node. Using microformats with CTS as a layer of indirection and invocation into implementation, the myriad Javascript widget libraries could agree upon a common calling convention that would be portable across implementations, freeing web developers from library lock in.

So too could this practice be applied to web themes, which are currently tightly coupled to vendor-specific content management systems. Or even to provide theming without needing a CMS at all. Consider the case of purchasing a web site template, a big business on the web. These for-sale templates come in a variety of formats: Photoshop documents, Flash files, HTML mockups, and custom CMS plugins. But today, these documents can not be used immediately: they first have to be chopped up and incorporated into some other templating language. Imagine instead if these templates were all delivered as HTML presentation mockups con-

⁵as does HTML5 Components [25], but using custom HTML tags for invocation

5. *Static Content Authoring*

forming to a microformat for their particular domain (photography website, mobile app advertisement website, blog, company page, etc). A blog template might have a microformat with classes like `article-list`, `article`, `title`, `author-name`, and `tag`. Any *content-laden* page on the the web could link to a tree sheet which imported that particular theme because they both made use of the same microformat.

We'll now discuss both of these use cases of CTS microformats in detail: widgets and themes.

5.2.2. **Widgets**

Many content authors want to take advantage of rich widget libraries, but their lack of Javascript knowledge prevents them from doing so. In interviews with web bloggers, we have found instances of people using relatively heavy-weight Javascript frameworks like Exhibit [48] just for the comparatively minuscule feature of sortable HTML tables. They reported choosing Exhibit because it offered them a way to accomplish this by just editing HTML, no Javascript necessary [8].

The D3 visualization library is another great example [15]. D3 provides the ability to create stunning web visualizations, and the project web site has a gallery of examples for people to copy, modify, and use on their own. The challenge is the difficulty of doing so: using D3 requires knowledge of Javascript, SVG, JSON, and D3's unique programming model. Even for an experienced Javascript programmer, this combination takes time to learn.

The HTML5 Specification's Web Component specification is one emerging standard for creating and reusing web widgets [25]. It enables custom HTML tags to expand into custom widget definitions.

This section shows how CTS can be used by widget authors to package and publish their widgets. The same implementation used to provide an example and documentation of the widget can also double as the production asset linked to and reused by widget consumers. We also show how widget consumers can make use of these widgets by including a tree sheet and then applying a CTS microformat to their page.

5.2.3. **CSS Shims as Simple Widgets**

A simple kind of widget is one that acts only to wrap a single HTML node and act as a shim for missing, but wanted, CSS functionality. A good example is vertically

5.2. Reusing Content and Presentation

centering an element. The `vertical-align` CSS property applies only to table cells, not block elements like `p` and `div`. But HTML authors often wish to vertically center content without regard for the node type in which the content resides. As a result, the web is filled with how-to guides containing tricks for vertically centering block elements. One such trick is to wrap the content in two containers---the outer one told (via CSS) to behave like a table and the inner is told to behave like a cell. But this trick requires presentational HTML scaffolding that must be applied over and over again; it can not be semantically associated and applied to elements merely by CSS classes.

Figure 5.6 show the code necessary to create and publish a vertical alignment widget using CTS. The HTML at left provides a working mockup of the widget that can be published and viewed both as a reference example and also the implementation. The CSS in the middle styles that mockup. And the tree sheet at right provides a mapping which creates a way to invoke the widget and map data into the widget using a microformat of CSS classes. The `@html` and `@css` lines at the top of the tree sheet reference the other two documents as entities in the same forest.

<u>example.org/widget.html</u>	<u>example.org/widget.css</u>	<u>example.org/widget.cts</u>
<pre><div id="vcenter-template"> <div class="t-wrapper"> <div class="c-wrapper"> I am a vertically centered widget. You can use me by applying the CSS class .vertically-center in your page after including my tree sheet. </div> </div> </div></pre>	<pre>.t-wrapper { display: table; } .c-wrapper { display: table-cell; vertical-align: middle; }</pre>	<pre>@html widget example.org/widget.html; @css example.org/widget.css; .vertically-center :graft widget #vcenter-template ; .vertically-center :is widget .c-wrapper ;</pre>

Figure 5.6.: Example of a CSS shim using CTS that vertically aligns any element with the `vertically-center` class.

Having provided this example implementation in HTML, CSS, and authored a tree sheet, this wrapping action can be applied to any element which contains the CSS class `vertically-center`. The HTML author need only include a link to the tree sheet in their web page:

```
<link rel="treesheet" href="http://path.to/widget.cts" />
```

And then apply the CSS class:

5. Static Content Authoring

```
<p class="vertically-center">
  I am vertically centered.
</p>
```

Figure 5.7 shows how the presence of the above resources in the CTS forest result in the proper application of the widget. The mockup is shown in blue, at left, and the invocation is shown in pink. HTML output by the CTS engine is shown below each step where output occurs.

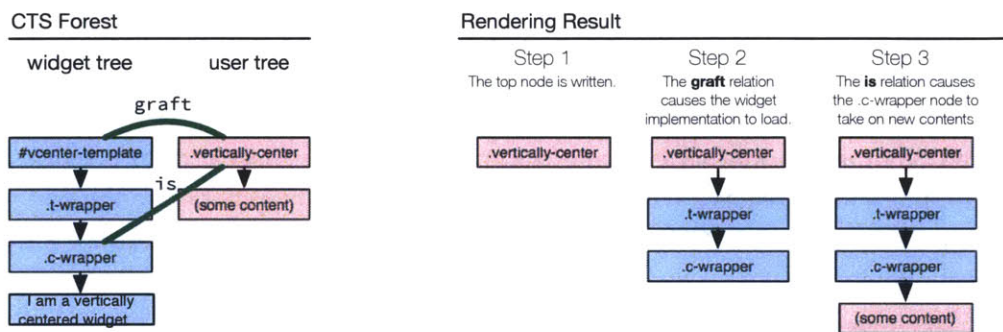


Figure 5.7.: Example of a CSS shim using CTS that vertically aligns any element with the `vertically-center` class.

By conservative estimates, using a CTS widget for the vertical align trick saves the author from having to create two DOM nodes *per centered element*, and by liberal estimates it saves the author from typing three CSS properties as well. It keeps the signal-to-noise of the novice's HTML high by limiting the amount of structural HTML required. But most importantly, it saves the user from having to remember or track down the appropriate trick for vertically centering text the next time around. Any time vertical centering is needed, this tree sheet can be linked from the page to provide the widget.

Widgets with Javascript and Complex Parameterization

CTS-based widgets can also make use of Javascript. As an example, we will look at a Javascript CTS widget made from a D3 choropleth visualization. First we will look at invoking the widget from the consumer's perspective. Then we will look at

what it takes to author and publish this widget. Finally we will examine the benefits of this practice across several D3 widgets.

Using a Choropleth Widget Let's say you are an elementary school teacher helping your class write a blog post about the population and industry of each state in the United States. Using the WYSIWYG editor in WordPress, you and your class create a table with two columns. In the first column is a list of state names, and in the second column resides the population of each state.

Now let's assume that a choropleth widget exists to help people make shaded maps of the US. To use this widget in your blog, you and your students perform the following steps:

1. First, you switch your editor out of WYSIWYG mode and into HTML mode.
2. Next, you add a link to the tree sheet that defines the microformat and mapping into the choropleth widget:

```
<link rel="treesheet" href="http://example.org/widgets/choropleth.cts" />
```

3. After visiting the widget's example page to read about the CSS classes its microformat defines, you then begin adding those classes to the HTML of your table:

```
<div class="us-choro">
  <table class="states">
    <tr><td>State</td><td>Population</td></tr>
    <tr><td>State</td><td>Population</td></tr>
    ...
```

In our reference implementation of this widget for this work, those are the only two classes required.

4. Finally, you return to the WYSIWYG mode and once again see just an ordinary table. But when you preview your blog post as an HTML page, you see a shaded map, based on your data, instead of the table.

This user experience required a bit of technical wrangling by the teacher. He had to add an HTML element and apply CSS classes to a few others. But we know from

5. Static Content Authoring

the Exhibit study that HTML-based widget invocation is something non-programmers can learn to do [7]. And think of all that the teacher did not have to do, such as understand Javascript, JSON, graphics libraries, or SVG. Once these CSS classes are added to the table, the class can continue editing the table in WYSIWYG mode and watching the changes in the shaded map that results.

Authoring a Choropleth Widget The author of this widget follows a set of steps to enable consumers to invoke, parameterize, and launch it's execution.

Invocation and parameterization are done with a tree sheet, like this one used in our reference implementation:

```
@html map http://treesheets.org/widgets/choropleth/index.html
.us-choro
  :graft map | #usa;
.us-choro .states tbody
  :are map | #usa .states;
.us-choro .states tbody td:first-child
  :is map | #usa .states .name;
.us-choro .states tbody td:last-child
  :is map | #usa .states .value;
```

This treesheet defines the CSS selectors describing the calling convention of the consumer and maps them on to HTML structure within the widget implementation. In this case, these structures within the widget implementation are nodes hidden by CSS, that act solely as a way to stash parameters in the HTML structure that can be addressable by CTS.

Given nothing else, this tree sheet, would load the widget's HTML implementation from `treesheets.org` (see the `@html` command at top), map the state and population data into an invisible DOM tree within it, and cause the table to disappear, replaced by only invisible content.

Next we need to get Javascript to run to render a shaded map inside the scope of the widget implementation. Widget authors can include a Javascript file from a tree sheet with the `@js` header statement. Just as the author might type

```
@html treeName http://path.to/some/html/file.html ;
```

They can type

5.2. Reusing Content and Presentation

```
@js http://path.to/some/javascript/file.js ;
```

to ask for the inclusion of a Javascript file when the tree sheet is loaded (an `@css` statement provides the same functionality for CSS).

So the next step for our hypothetical choropleth widget author is to include his Javascript and CSS files from the tree sheet. They will be included on the page whenever a consumer links to that tree sheet.

The only final step is making sure that the widget's initialization function is called and provided with the DOM node that represents the widget's root element on the page. This is a bit of a tricky area for CTS. The HTML5 Component Specification, since it takes liberties adding to the HTML specification, is able to define a strong model of scoping and encapsulation for widget Javascript.

Because CTS relies on only those capabilities available from Javascript, conventions must be used to invoke and pass the proper parameters to widget Javascript:

- First, write any widget initialization code as a function that takes the root element of the widget's HTML representation as an argument.
- Second, attach a listener on the DOM for elements with a special CSS class added to the root element in the widget presentation mockup. Since the tree sheet is co-developed with the widget, choosing this CSS class is something that the developer would already have had to do.
- When this listener fires, call the initialization code, passing the element responsible for the event.

This ensures that the initialization code is only run with a new widget instance is invoked (added to the DOM) and that it is passed the context of that invocation in the DOM.

Figure 5.8 shows an example of the Choropleth widget we've been discussing. The invoker's `table` element containing data is shown at left. The widget author's tree sheet, is shown in center. This maps the invoker's HTML into the widget implementation. A screen shot of the completed result is shown at right.

The Choropleth widget in Figure 5.8 shows how the relational approach used by CTS enables plain HTML to act as a calling convention into these complex widgets, making them accessible to novices. The HTML the novice writes is not just content to be displayed, but also parameters to pass to the widget, such as pin locations for a map or data for a bar chart.

5. Static Content Authoring

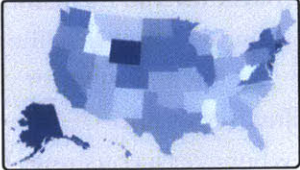
HTML	Tree Sheet	Widget Result
<pre><div class="us-choro"> <table class="states"> <tbody> <tr> <td>Alabama</td><td>36</td> </tr> <tr> <td>Alaska</td><td>65</td> </tr> ... (continued) ... </tbody> </table> </div></pre>	<pre>@html map http://treesheets..; .us-choro :graft map #usa; .us-choro .states tbody :are map #usa .states; .us-choro .states tbody td:first-child :is map #usa .states .name; .us-choro .states tbody td:last-child :is map #usa .states .value;</pre>	

Figure 5.8.: A choropleth widget, implemented with D3 and SVG. Users can invoke this widget with custom data by annotating HTML (at left) with the proper classes; no Javascript is necessary. The center column shows the CTS sheet which maps the user's HTML onto the widget for conversion into an SVG object.

Several Widgets Evaluated We implemented several widgets to evaluate this approach. The HTML to invoke these widgets can be entered by hand, while source editing, or via a CMS editing interface (our tests were done with the Wordpress post editor). The tree sheets we wrote for these widgets are designed to put all parameters into the visual space of the DOM, rather than into the attribute space, further enhancing editability in WYSIWYG environments like Wordpress.

Table 5.3 shows the effort savings from the perspective of the user. For each widget we created, we list the skills that a user otherwise would have needed to know to reuse the widget as published (at time of writing) on the D3 example page. In most cases, reusing these widgets requires detailed inspection of code in the baseline case, but simple copying and pasting of HTML in the CTS case. We additionally list the lines of Javascript that the user would have had to either write or modify to create that widget without CTS⁶.

5.2.4. Themes and Retargeting

Page-wide themes are simply page-sized widgets that *wrap around* a content document instead of embedding themselves within it. From that perspective, themes are just another type of CTS Microformat. Because of that, rather than discussing the creation of themes from the perspective of a theme author, we focus on a more inter-

⁶Lines of code are, of course, only useful as approximate measures of complexity, as this measure is affected by factors such as coding style and library usage.

5.2. Reusing Content and Presentation

Widget	SVG	JS	JSON	Ajax	LoC JS
Google Map		✓			64
Stock Ticker		✓	✓	✓	32
Choropleth	✓	✓	✓	✓	51
Bubble Chart	✓	✓	✓	✓	53
Word Cloud	✓	✓		✓	112
Bar Chart	✓	✓		✓	101

Table 5.3.: Skills that CTS saves the web author from needing to know for each widget we created. We also list the lines of Javascript that the author would have needed write (or modify) to create that widget without CTS.

esting use case: *post hoc* theming by ordinary web authors who did not initially set out to explicitly use or create themes. We will continue using the term microformat because the underlying principle is the same, even though we are now discussing page-level design.w

Any existing web page can be thought of as an HTML mockup, even if it wasn't designed with CTS, MDD, or Microformats in mind. Consider the case of a group of professors who all maintain personal pages. If one professor with a relatively spartan page wants to make his web page look like another professor's fancy page, this remapping can be performed by simply authoring a tree sheet which, semantic item by semantic item, relates objects on one page to objects on the other.

If each object (e.g., lab name, phone number, job title) is distinctly contained within a single addressable element on both pages, and list items are expressed as individual DOM nodes beneath a container node, then only this tree sheet is necessary to perform the remapping. If that is not the case, the HTML of one page or the other will need to be altered so that each distinct semantic entity can be uniquely addressed using a CSS selector.

While this approach works fine between any two professors, it is clearly not sustainable for all professors as it would require n^2 tree sheets, one for every pair of professor pages. Instead, we can imagine that a group of academics gets together to create a CTS Microformat for academic home pages, with CSS classes for all the concepts in common to the domain: lab name, office hours, publication list, conference titles and so on. This approach still allows n^2 mappings between pages, but

5. Static Content Authoring

via a single unified tree sheet (the microformat). In spirit, it is an approach similar to the hub-and-spoke design of federated database systems [80].

This theme microformat is encoded as two assets published to the web. First, a tree sheet which maps each CSS class in a content document to a presentation document. For example, it might contain the following fragment:

```
content | .pub-list      :if-exist theme | .pub-list-wrapper ;
content | .pub-list      :are       theme | .pub-list ;
content | .pub-title     :is        theme | .pub-title;
content | .pub-authors   :are       theme | .pub-authors;
content | .pub-year      :is        theme | .pub-year;
```

...and so on.

Note two particulars about the construction of this tree sheet. First, the content and theme document are unbound. Binding these tree labels to actual HTML trees is left to the consumer of the tree sheet. Second, the same selectors are used for both the content document and the presentation document. This means that any web page which applies these CSS classes can both make use of themes and also serve as a theme themselves for other pages. This allows anyone with a content document conforming to those selectors to retarget their page as any other web document conforming to the same selectors. Importing this tree sheet and binding the two trees would be done as follows in the head:

```
<link rel="treesheet" href="http://path.to/academic.cts">
<style type="text/cts">
  @html content this;
  @html theme http://fancy-professor.com/index.html ;
</style>
```

The second asset the microformat community would create is a bare-bones *exemplar page*. This example page is an *optional* content document that contains an example of invoking all the elements of the design mockup. Rendered in a web browser, it would look like an unadorned 1993-style web page, with some mockup data and perhaps a small amount of design scaffolding to mark major sections in the document. This example page acts as a good start place for new professors, who can copy it and replace the mockup values with their own information.

5.2. Reusing Content and Presentation

If a professor already has a site and does not want to copy the exemplar page and start from scratch, he could modify his page to make use of the CSS classes in the microformat (e.g., `.labName` and `.officeHours`). Like many web authors, this professor did not originally surround all his content with DOM elements: some pieces of information are entered as raw text with line breaks (`
`). In these cases, the professor has to first wrap the content in a `div` or `span` tag before adding the CSS class.

After adding the proper CSS classes, he can re-theme his page by (1) linking to the CTS sheet that binds the microformat he just added to a design document, and (2) binding the design document placeholder to an actual design mockup, as with the code example above. Because he uses the CSS classes in the microformat, **his page now also acts as a design mockup**, meaning other academics can remap their content onto his design. This turns every CTS-using web author into a theme creator, a powerful change from existing practice.

Themes and widgets (and more generally, CTS transformations) can also be combined, composed, and nested in different ways. Perhaps there is a nice CTS widget that shows a contact list for a set of items using the vCard microformat. The professor could include this tree sheet and then apply those vCard CSS classes to each of his students' information. The student list would be stylized by the widget and still remain located within the broader style provided by the theme.

We evaluate this idea by creating a microformat for the academic domain and then modifying a handful of professors' academic home pages to conform to this microformat. We mapped domain concepts (such as *job title*) instead of design concepts (such as *sidebar*), though the latter would be an alternate route. Table 5.4 shows the number of HTML additions necessary for each of these four professor pages to use this microformat. In all cases, there existed separate concepts in the microformat that were commingled together in HTML as text within a single DOM node. One addition then, as we counted, was the creation of a new DOM node to uniquely wrap a single semantic entity described by the microformat.

Figure 5.9 shows the output of a test harness that enables browsing the n^2 mapping combinations possible once several pages are all adapted to make use of the same CTS microformat. The results demonstrate an interesting challenge: the information on a professor's landing page can vary wildly. Some showcase links and students; others showcase papers and news. When mapping at such a low level,

5. Static Content Authoring

Professor	Karger	Yardi	Landay	Kolko
Element Additions	2	11	5	6

Table 5.4.: New elements required to map four professor pages found on the web into an example academic microformat we created. The result enables the retargeting shown in Figure 5.9.

this can result in both gaps in the resulting page (which expected information that it did not get) and missing content from the source (because there was nowhere to put it).

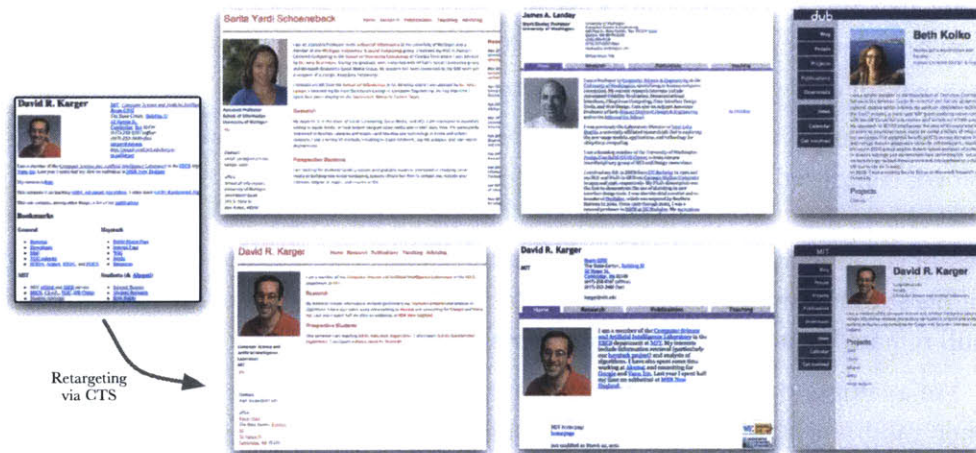


Figure 5.9.: Some remapping possibilities after creating a four tree sheets, relating different professor pages to a common set of selectors. Not every page contains the same semantic information, but the information in common (e.g., name, bio) can be mapped while information lacking (e.g., news updates) is hidden by the retargeting process.

Practically, this means that if a tree sheet is to be used with a microformat community such as this one, some elements on a page will require two CTS rules instead of one to be robust against missing information. In particular, if an element has decoration surrounding it, it may need an `is` relation to map content inside of it, and an `if-exist` relation to prevent the surrounding decoration from appearing if

5.2. Reusing Content and Presentation

that content is missing. An example visible in Figure 5.9 is the mapping from David Karger to Sarita Yardi. David does not have news items on his page, so in addition to not displaying any news items (handled by the `is` command), the news section header should also be hidden (by the `if-exist`).

Schema Alignment. The problem of missing data also highlights the more general schema alignment problem that is to be found any time two different data sources are being coordinated. No matter how well-structured the data authoring practice between two individuals is, odds are they describe data from the same domain slightly differently, requiring a translation step in order to coordinate data. Any one instance of this translation step is trivial, but non-triviality comes from the fact each new member of the ecosystem brings yet another slightly variant idea of schema—the problem is never “solved” given a growing community, though one would expect the presence of a microformat to encourage schema convergence.

In the case of CTS, here is how someone seeking to retarget a page would handle the problem:

- When a design styles content that you don't have, simply create an `if-exist` relation between that region of design and some selector on your content page that matches the null set of nodes. When the final page renders, that design element will not appear.
- When you have content that a design mockup doesn't style, this is a bigger problem. You could find an unused portion of that page and use an `is` relation to shove your content in there, but that might not always be a realistic option.

For this latter case, and even in the former, manually copying the design and then modifying it may be necessary.

Machine learning approaches like Bricolage [61] may prove useful in solving the schema alignment problem as it relates to web content and design. As may visual approaches to annotating semantically like (or, in a pinch, visually compatible) elements between pages, such as Dontcheva's work with visual web scraping and merging [28]. In particular, it may help in finding the sweet spot between layout-centric mapping (headers, sidebars, navs, etc) and domain-centric data mapping (office address, name, title, etc).

5. Static Content Authoring

Building a long tail of theme domains

Importantly, there is no need for a critical mass to benefit from CTS-based theming--even the first professor to use CTS benefits from the separation of presentation and content, as shown in the authoring section. However, there is a network effect offering greater benefit (more themes) as the number of participants grows.

This method of theming could dramatically expand our cultural ideas of *when we might use themes*. Our current conception of themes is bound to the idea of content management systems, and theme languages are intimately tied to their implementations. Wordpress, Drupal, Blogger, and Tumblr all have roughly the same underlying schema, but a theme written for any one is completely incompatible with the others. This creates two problems: it limits "theme creators" to a specific set of developers who set out specifically to write themes for a particular platform, and it requires committing to a complete server-side architecture in order to use these themes. Could it be that there are no themes for the "professor home page" domain in part because there are no content management systems built for this domain?

The answer is not simply to use client-side templating languages, such as Ember [86] and Mustache [92]. These languages require JSON as a data input, and JSON is a poor format for content authoring due to (among other issues) its lack of support for multi-line strings [7]. They also describe transformations instead of tree relations; they can't consume a finished web page, *after the fact*, as input for a remapping operation, or compose multiple relations.

CTS, MDD, and microformats enable theming ecosystems to arise after the fact for arbitrary domains and without the need for content management systems. Mobile app landing pages, art galleries, auctions, yard sales, and academic information are all domains with sizable web user populations but no CMS or theming support. These domains do have many mockups, however, available as live pages on the web or for sale on sites like `TemplateMonster.com`. These design examples could be used and reused as themes *by reference* instead of requiring them to be cut up and transliterated into a template language *by value*.

5.3. Maintaining Code: Proper Encapsulation

In this final section on static authoring, we turn to code maintenance. Skilled programmers intuitively feel that proper encapsulation is an important property that

5.3. *Maintaining Code: Proper Encapsulation*

facilitates software maintenance and development. This intuition is backed up by studies that have shown that encapsulating and layering software components results in higher productivity [97]. Over time, bugs are identified and fixed, improvements are made, and sometimes implementation is just rearranged for legibility. Code that is well encapsulated can undergo such changes without affecting downstream consumers of that code. Here we look at two aspects of encapsulation: the ability to shield a code consumer from implementation particulars and from implementation changes.

Any piece of reusable web content can be thought of as a module that mixes three different languages: CSS, Javascript, and HTML. Today, consumers of such modules can link to CSS and Javascript by reference, but the concept of "linking to HTML" is absent from web development. Instead, module consumers must copy the HTML implementation by value and then adapt it so that the content particulars reflect the consumer's content rather than the example's mockup content. This manner of reusing web content is thus a bit like having to copy and paste one third of the C Standard Libraries despite being able to dynamically link to the other two thirds.

More importantly, this practice creates an encapsulation leak: these HTML structures are part of the implementation, but they are copied by value out of the library and into every user's web page. Between libraries, and sometimes between different versions of the same library, these HTML implementations are incompatible with other.

Libraries of reusable web presentation such as Twitter Bootstrap suffer from this encapsulation leak. As just one example, when the Bootstrap library was upgraded from version 1 to version 2, the CSS and Javascript implementations changed significantly enough to require a change in the paired HTML structures as well. Any site maintainers who did not both know the details of these changes and then also manually apply them across all their HTML documents risked breaking their page layouts.

Figure 5.10 shows one example of how pages might break if they used Bootstrap's horizontal form widget. Because the HTML implementation for this widget changed, linking to Version 2 of the CSS and Javascript while continuing to use Version 1 of the HTML implementation resulted in an incorrect layout.

Relational web authoring with CTS solves both these problems. It provides a

5. Static Content Authoring

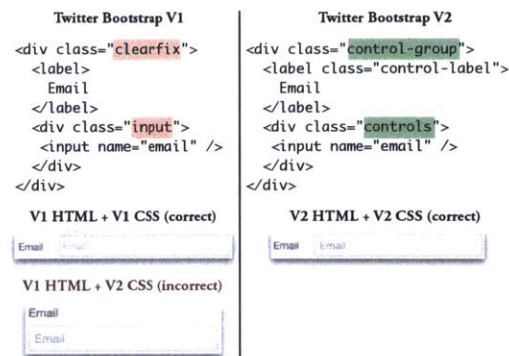


Figure 5.10.: The HTML implementation for Twitter Bootstrap's horizontal form widget changed between between version 1 and 2. While users could upgrade their Javascript and CSS by simply relinking files, this HTML change had to be made by hand. If it wasn't made, the form layout was broken.

layer of indirection so that the caller's HTML is merely an invocation into the widget's HTML structure, not the actual implementation. And it provides the ability to link to this transformation so that the latest version may always be applied. It therefore provides a way for widget developers to provide better quality libraries to their users.

To solve the Bootstrap problem with CTS, the library maintainers would define an HTML calling convention for each widget and publish a tree sheet that mapped these calling conventions into the current HTML implementation hosted on their own web site. Users of the library would then link to the tree sheet, which would in turn have references to any relevant HTML, CSS, and Javascript resources. When changes were made to the HTML implementation, changes would also be made to the relations in the tree sheet to reflect them.

Even if CTS was introduced late in the process, along with Version 2 of Twitter Bootstrap, it could be used to roll forward old HTML calling conventions like so:

```
@html twitter www.twitter.com/bootstrap/widget.html;
```

```
div.clearfix {attribute: class}
:is
twitter | div.control-group { attribute: class }
;
```

5.3. *Maintaining Code: Proper Encapsulation*

```
div.input {attribute: class}  
:is  
twitter | div.controls { attribute: class}  
;
```

These rules provide a "hot fix" that converts Figure 5.10's Version 1 HTML (the target selectors) to Version 2 HTML (the source selectors), ensuring that the rendering does not break due to mismatched HTML and CSS.

6. Reactive Programming, Data, and Computation

Creating a web page that accepts and saves data from user, computes upon that data, and shows dynamic responses requires both programming knowledge and system administration skills. As consumers of the web, we have grown accustomed to browsing sites with this type of functionality. But as authors on the web, we are often hampered by how much effort is involved in creating these kinds of dynamic sites. The level of effort is high enough to place it outside the scope of what many web authors can build at all. And even for experienced programmers, significant time and labor is required.

The fact that the form-building application called Google Forms is so popular is a testament to this problem. Google Forms does nothing other than append rows to a spreadsheet. But this basic append-only input operation is so labor intensive to build without a service like Google Forms that beginners and experts alike would rather use Google Forms (which do not permit custom designs) than create their own data collection pages.

As the degree of interactivity increases, so does the amount of intellectual scaffolding and labor required to build the page. A website called TodoMVC acts as a Rosetta Stone for web-based todo applications, the Hello World of web programming. A recent trend in web programming frameworks visible on this site is so-called "Reactive Programming," which seeks to hide state replication (between the client, app server, and database) from the programmer. While these applications provide interesting architectural models for desktop-style application programming on the web, looking at the code on TodoMVC.com reveals that these programming paradigms require even something as simple as a todo list to require significant Javascript coding to create models, controllers, event handlers, and sometimes Javascript-based views.

This chapter shows a radically different technique using CTS that achieves much

6. *Reactive Programming, Data, and Computation*

of the same functionality but without any programming. Because CTS relations are lossless (Section D), the relations used to compose the mix of data, content, and design that forms a web page are preserved as annotations atop the final output. And because CTS relations are symmetric (Section 4.2), these relations can be used to synchronize changes the web reader makes to the page with the data structures that composed the page. These two properties provide the basis for a CTS-based runtime system called Quilt which endows web pages with simple create, read, update, delete (CRUD) capabilities using nothing other than the tree sheets used to display that page in the first place [11].

Using only HTML and JSON trees would result in pages with simple information management capability but no data management system or computation capabilities, so Quilt additionally adds support for connecting to Google Spreadsheets as if they were a tree. With a spreadsheet as an addressable entity in the CTS forest, a change in the web page can trigger a change in the spreadsheet, which triggers a computation, which updates a cell, which triggers a change back in the web page.

The category of application Quilt is capable of enabling CRUD applications with simple computation, global and private data scopes, and only basic data manipulation (for searching, sorting, etc). This is a simple category of application compared to modern Javascript frameworks capable of arbitrary functionality, but it is also in incredibly important one. Even complex web applications like Twitter and Facebook still have—as the mainstay of their browser-server interaction—a major CRUD+Compute component.

For novice authors, Quilt represents both an authoring environment and an execution platform that makes new categories of application authoring previously impossible to them. Expert authors love an easy solution as well, of course (many use Google Forms instead of write custom code, for example). And for simple applications Quilt offers a faster way to accomplish what they might have otherwise have written as code. For more complex applications that need to scale, Quilt represents an interesting authoring environment that might, by some future work, be compiled into a runtime system that scales more than spreadsheet software would be expected to in production.

We this chapter first discusses the topic of spreadsheet-backed programming, then describes the modifications to CTS that we made to create the Quilt system. Next then report results from a user study of 15 end users with basic HTML editing

6.1. Spreadsheet Backed Web Applications

ability. After learning our method for fifteen minutes, they performed a series of authoring tasks in which they connected spreadsheets to web pages when provided both and authored (and then connected) spreadsheets from scratch when provided only an HTML UI mockup. Participants were nearly all able to complete the tasks successfully in only a few minutes. We use videos, surveys, and interviews from this user study to evaluate Quilt's learn-ability and reflect on debugging with this method of authoring.

6.1. Spreadsheet Backed Web Applications

Spreadsheets are ubiquitous information management tools that cross technical and cultural divides. College students use them to plan parties, financial analysts use them to sway markets, and many companies use spreadsheets instead of databases to manage data [90]. According to the 2012 United States Census, over 55 million people in the United States "manipulate data with either spreadsheets or databases at work [77]." But while these authors use spreadsheets for a diversity of information management tasks, they have limited ability to view, edit, and share their data outside the grid-based interface spreadsheets provide. Showing a stylized schedule for an event planned with a spreadsheet, for example, typically requires copying and pasting raw data from the spreadsheet into the finalized design document.

There is a separate overlapping community of people who know how to edit HTML. The size of this community is difficult to assess, but web literacy is undoubtedly growing. The ACM Computer Science Standards cite HTML and CSS authoring as basic components of computer literacy at the K-12 level [2]. Web authors build pages for a variety of purposes, ranging from personal to public to collaborative. Like spreadsheet authors, web authors face a steep hurdle once they try to move beyond the basic capabilities HTML offers. To create a web page with data storage and computation, an HTML author must learn to write client-server programs and possibly to design and administer databases.

In both cases, users have access to a tool that is specialized for one type of information-centric activity, such as data management or hypertext publishing, but does not natively support other activities. Past approaches to these two problems either focus heavily on providing pre-made widgets or templates [48] [88] or use new authoring and deployment environments that replace the user's existing ones [95] [74].

6. Reactive Programming, Data, and Computation

From the perspective of end-user web programming, Spreadsheets and HTML appear to be complimentary technologies waiting to be joined. Spreadsheet users enjoy structured data and computation but lack the ability to view and edit data in whatever flexible, stylized fashion suits them. HTML authors can create such styled displays, but studies show they both lack and greatly desire the ability to perform information management tasks such as data collection, storage, and retrieval [74] [75] [7], exactly what spreadsheets can step in to provide.

Spreadsheets as Models with Debugging Interfaces One way to think about spreadsheet-backed applications as a programming pattern is to think of the spreadsheet as a model with a debugging interface attached. The common Model-View programming pattern does not make any statements about the role of the model, apart from its role as the broker of data. Spreadsheets add to this role all of the actual spreadsheet software and functionality that comes with it, turning it from programmatic concept into one that is also user-facing for both the developer and the end-user.

This means that all data and computation is visible, editable, and debugable in a well known environment, not hidden behind controller code and database APIs. It also brings data modeling and computation underneath a uniform umbrella (the spreadsheet) that has been well studied as an end-user programming environment [68] [67] [18] and a debugging environment [60] [56] [6]. Many bugs can be identified and resolved entirely as spreadsheet issues, independent of the HTML interface, or vice versa.

6.2. An Example: A Spreadsheet-backed Microformat Widgets

As a motivating example, think back to the widgets from Section 5.2.2. They could be parameterized and import Javascript functionality, but they were all read-only.

Imagine instead a widget encapsulated by a tree sheet which bound input boxes on a form to a spreadsheet that collected up to three fields of information. From the widget consumer's perspective, invoking this widget would only require applying the CSS classes it defined:

1. `.collect` to the FORM element

6.3. Incorporating Spreadsheets into CTS for End-User Programming

2. `.field1` to the first INPUT element to save
3. `.field2` to the second INPUT element to save, and
4. `.field3` to the third INPUT element to save

By importing the tree sheet defining this microformat and then binding the spreadsheet tree placeholder name to a *particular* spreadsheet, the widget consumer could now decorate any number of HTML mockups on her site and those mockups would append rows to the spreadsheet. That user might then go one step further and compose widgets, using yet another blog theme widget to stylize these three fields as the comment box for each blog post.

If data flows in both directions, any number of read-write-compute widgets could be created by authors, published for reuse via tree sheets, and then invoked by consumers via CTS microformats.

6.3. Incorporating Spreadsheets into CTS for End-User Programming

This Quilt work was done in the context of an end-user programming study. Our goal was not just to incorporate spreadsheets into the CTS paradigm but also to simplify it as much as possible for easy use by HTML novices. This section details the three steps we took to integrate spreadsheets toward that goal: adding a new `SpreadsheetTree` type, simplifying the CTS language, and creating a selector language for spreadsheets.

6.3.1. Casting Spreadsheets as Trees

To fit spreadsheets into a form that works with the CTS engine, we simply cast it as a tree and design a selector language to allow developers to select different elements of the spreadsheet. From an implementation point of view, this tree implementation is just

Abstractly tree representation used for Google Spreadsheets redundantly projects each workbook in a spreadsheet multiple times for different styles of access. Figure 6.1 shows a partial example. The top level represents the entire spreadsheet document, followed by a level of nodes representing each worksheet in the document.

6. Reactive Programming, Data, and Computation

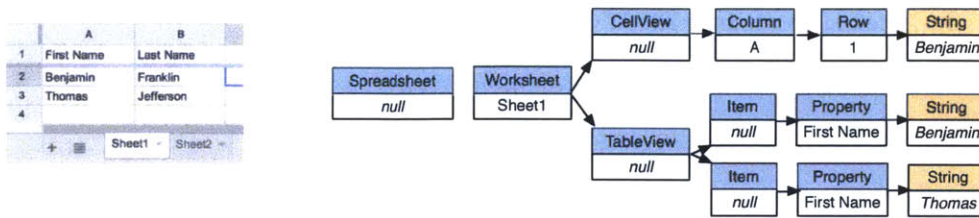


Figure 6.1.: An example depicting how CTS interprets a Spreadsheet as a tree. The tree shown here is only a partial representation of the depicted spreadsheet for space reasons. Note how the same data appears projected in different ways, enabling different style of addressing the data.

The third level contains nodes that represent various projections of the spreadsheet, in this case `CellView` and `TableView`.

The `CellView` is a projection useful for accessing individual cells in the spreadsheet. The `TableView` is useful for addressing the spreadsheet as a row-oriented table of items. It assumes that a header row provides the names of the attributes that each item has. Each non-header row represents one item, and each column represents the cut of this attribute across all items.

While the CTS prototype only contains these two projections, many other useful projections are possible. A column-oriented table is one possibility. Another would be a parameter-list style projection that searches a spreadsheet for blocks of cells that appear to be key-valued pairs. For example, the inputs to a mortgage calculation. The projection would expose the values (data cells) by enabling the CTS author to address them using the keys (label cells).

6.3.2. Simplifying the Language

Our next adaptation was a simplification for the sake of end-user programmability: cutting away as much surface-level syntax as possible.

Since spreadsheets are strongly structured data objects, it is always clear whether a particular spreadsheet entity (e.g., the list of rows, cell A1, a particular column) acts as a set of values or a single value. Since this is exactly the semantic difference between `is` and `are`, we combined them into a new relation `connect` and adapted the CTS engine to translate this relation into `is` or `are` based on runtime context.

We then modified the CTS parser to look for CTS statements directly authored as

6.3. Incorporating Spreadsheets into CTS for End-User Programming

the HTML attributes `connect`, `show-if`, and `hide-if` rather than the ordinary full-length, semicolon terminated CTS statements in a `data-cts` attribute. While teaching CTS authoring this way comes at the expense of the re-usability external tree sheets afford, it relieves authors of having to understand CSS selectors and further makes the appearance of CTS commands in the HTML plain looking in comparison to "standard CTS."

Finally, we removed the need to specify the name of the spreadsheet tree in selectors that occur within these new HTML attributes. If absent, the default tree name "sheet" is assumed. These modifications, shown in Figure 6.2 are an extension layered on top of the standard CTS engine. The author can fall back on normal CTS at any time.

<u>External Tree Sheet</u>	<u>Inlined Tree Sheet</u>	<u>Quilt Variant</u>
HTML	HTML	HTML
<code><div id="#price"></div></code>	<code><div data-cts="this :is sheet A1;"></div></code>	<code><div connect="sheet A1"></div></code>
CTS		
<code>#price :is sheet A1;</code>		

Figure 6.2.: Results of simplifying the CTS syntax for ease of use by non-programmers. The External and Inlined Tree Sheet methods are the two ordinary ways the CTS Engine accepts statements from which to build relations. The Quilt variant simplifies this and relies on the structure of the spreadsheet to fill in the missing user-specified information.

6.3.3. Spreadsheet Selector Language

The final adaptation concerned addressing the spreadsheet using a selector language, which was done using a slightly expanded variant of the language spreadsheets already use internally. This section provides an introduction to that language similar to the one provided to participants in our user study of Quilt.

Assume a sample task of building a web interface to collaboratively plan a party with friends. The following spreadsheet contains a list of food items, their costs, the person responsible, and a computed cell for total cost:

6. Reactive Programming, Data, and Computation

	A	B	C	D
1	Item	Cost	Person	
2	Cheese	\$15.00	Casey	total
3	Crackers	\$5.00	?	\$35.00
4	Wine	\$15.00	?	

Quilt reactively connects an HTML web interface to the spreadsheet using four new HTML attributes: Depending on where they are used, these attributes can synchronize data between the web page and spreadsheet, add new rows, delete rows, and show /hide web elements depending on spreadsheet contents.

The following code snippet and output demonstrates how to connect HTML elements to spreadsheet cells. Cells can contain either plain text or computed values. In both cases, changes made on the spreadsheet flow automatically to the web page, replacing the element content. Changes to the web element content, through direct user action or Javascript manipulation, propagate automatically back to the spreadsheet.

```
<span connect="B3">cost</span> → $35.00
```

By placing the attribute `connect="rows"`, Quilt users can designate that an HTML element's contents should repeat once per row in the spreadsheet. Each repetition corresponds to the data in a row. Inside a repetition, one can connect HTML elements using the names of columns, or offer row deletion by placing `delete="row"` on a button. As before, runtime modifications, insertions, or deletions are synchronized between the web page and the spreadsheet.

```
<div connect="rows">
  <div class="party-item">
    <span connect="Item">item</span>
    <button delete="row">x</button>
  </div>
</div>
```

→ Cheese ☒
Crackers ☒
Wine ☒

When a form carries the attribute `connect="rows"`, submitting that form appends a new row. Input elements connected to column names provide the new values for that row. When the new row is appended to the spreadsheet, data flows back to any connected elements on the web page.

```
<form connect="rows">
  Item: <input connect="Item">
  Cost: <input connect="Cost">
  <button>Add</button>
</form>
```

Item:
Cost:

6.4. Synchronizing Data as it Changes

Finally, the commands `hide-if` and `show-if` hide and show a web element based on whether the connected cell is “Truthy,” represented by anything other than an empty cell, the string `FALSE`, or the number 0. This example limits the output to show only food items with no assigned person:

```
<div connect="rows">
  <div class="party-item" hide-if="Person"> ... </div>
</div>
```

Finally, the spreadsheet standard `WorksheetName !` prefix can begin a selector to denote it only applies to a particular worksheet in the spreadsheet. Without such a prefix, the selectors applies to the union of its match across all worksheets.

Using these annotations, the Quilt engine can turn a static web mockup and a spreadsheet into a simple read-write web app, with the spreadsheet adding computation as well.

6.4. Synchronizing Data as it Changes

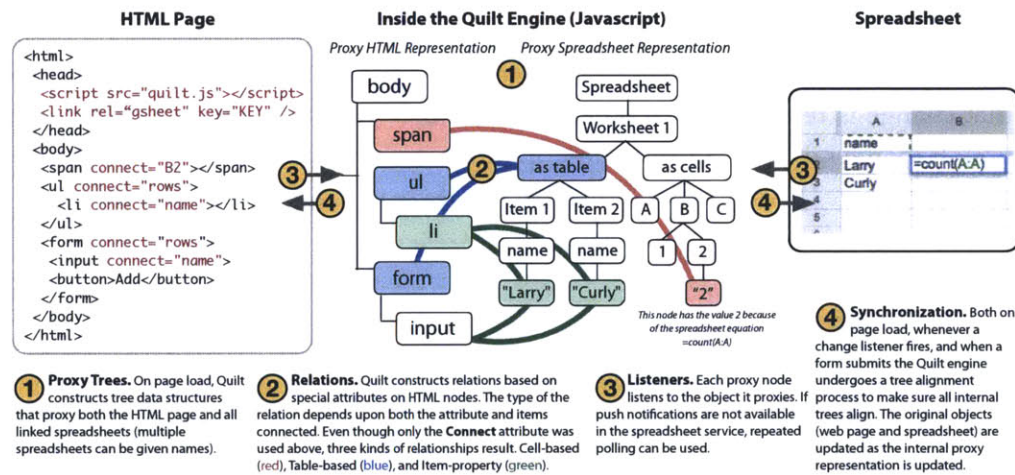


Figure 6.3.: Quilt maintains mirrored, tree-shaped representations of the HTML and spreadsheet. It aligns these representations, rewriting the source objects whenever synchronization is appropriate, such as on web page load or user input.

This section describes the underlying changes to the CTS architecture that enables it to serve as a reactive read-write engine in addition to a rendering engine. A

6. Reactive Programming, Data, and Computation

condensed diagram of the whole process is provided in Figure 6.3.

Quilt requires little additional architecture on top of the CTS rendering engine. In fact, in at least the forward rendering direction (from data source to output page), because the output is pure function of the inputs, the rendering process could simply be performed once again entirely whenever an input changes. But because we would like data to travel in both directions, and because re-rendering the entire page whenever a fragment changes would not be efficient, we'll develop an event-based synchronization method here.

Recall that a web page built with CTS is the output of the rendering process defined in brief in Section 4.6 and in detail in Appendix B

$$\text{render}(\text{root}, \text{trees}, \text{forrest}) : t_{n+1}$$

This output tree, t_{n+1} , is the document shown in the web browser. It contains clones of all the relations that were utilized to make decisions about its construction.

The Quilt engine handles synchronization in a manner modeled after the Operational Transform literature [29]. Quilt attaches change listeners to every node in each tree in the forest. In the Quilt scheme, each tree represents what the OT literature typically calls a user. When a change event occurs, Quilt determines whether that change is relevant to the other trees. If so, it generates a transform description. It then sends the transform description to the other tree, which applies it to its own structure. This section describes each of these two main processes: generating transforms and applying transforms.

6.4.1. Generating Transforms

CTS trees might represent HTML, JSON, Spreadsheets, or any other data object. From an implementation perspective, each has a different implementation of the tree node objects that is sensitive to the underlying native data structure and the way changes in it correspond to insertions, deletions, and modifications in the tree. This way, the Quilt engine does not need to understand the details about the real data structure (e.g., spreadsheet) behind the tree.

The Quilt engine listens for node insertions, deletions, and modifications on the trees in its forest. When one of these changes occurs, the next task is to decide whether it is relevant to the other trees. Relevancy, here, is a function of where that change occurs with respect to the relations on the tree:

6.4. Synchronizing Data as it Changes

- If any change happens with the subtree of an *is* -related node, that change represents an *Update* operation to that *is* -related node.
- If the direct child of an *are* -related node is deleted or inserted, that change represents either a *Create* or *Delete* operation on that *are* related node (or rather, the set implied by that *are* -related node).

Starting with the node the event was thrown on, Quilt walks the tree toward the root looking for the first *is* or *are* relation it comes into contact with, after which the event bubble is halted. If, at that node, one of the above two conditions are met, a transform object is created describing the transform that occurred from the perspective of that node.

Quilt only supports the three transform types above: *Update*, *Create*, and *Delete*. But together with the initial render, which represents the *Read* operation, this completes the create, read, update, delete set of CRUD operations.

Once created, the transform is passed along the relation to the related node on the other side. To return to the OT metaphor, this is equivalent to passing an operational transform to another user, the payload of which specifies the type of action that occurred, the arguments necessary to describe that action type, and the location in the document (the related tree node) where it occurred.

6.4.2. Applying Transforms

When a node receives a transform object from a related node, it interprets that as a request to modify the structure of its subtree somehow:

- If the relation was an *is* and the transform was an *Update*, the subtree should be rewritten with the new value contained.
- If the relation was an *are* and the transform was a *Delete*, a child node should be deleted. Which child is specified in the parameters of the transform.
- If the relation was an *are* and the transform was a *Create*, a new child node should be created. Which index to insert this new child is specified in the parameters of the transform.

The first two cases are self-evident in their implementation, but one important detail is worth mentioning. Different node types might interpret *Update* values in

6. Reactive Programming, Data, and Computation

different ways. For example, the CTS tree-node wrapper for an HTML Checkbox interprets a truey value as the checked state and anything else as an unchecked state, whereas other HTML node wrappers simply set their `innerHTML` property to whatever the *Update* contained.

The *Create* operation requires a bit more detail. Think back to how CTS renders an `are` relation: it involves copying a subtree, aligning relations within the subtree to ensure that it is scoped to the i^{th} subtree of the related node, and then executing the CTS engine on that subtree. Since Quilt mutates trees rather than simply re-rendering them completely, it will have to apply this process just to the new node inserted. For example, it will clone the last child of the `are` related node, re-align all the relations of that child to match the new subtree on the other side of the `are`, and then run the CTS engine on it. This operation represents the difference between the tree before the transform and what the tree would have been if Quilt were to have re-run the rendering engine completely.

6.4.3. An Example Sequence of Transformations

Consider the following Todo list, annotated with Quilt-style CTS commands to connect this HTML fragment to a spreadsheet¹.

```
<ul connect="rows">
  <li>
    <input type="checkbox" connect="done" checked />
    <span connect="task-name">Defend Thesis</span>
  </li>
</ul>
```

If a user injected a new `` element with a checked checkbox and a span with the value `Eat Cake`, the following sequence of transforms would be sent:

1. *Create*(1) would be sent to node connected to the `ul` because of its `are` relation (written here as a `connect` statement to the spreadsheet `rows`). That would cause the spreadsheet to duplicate the 0^{th} row and add it as the 1^{st} row. The

¹Here, `connect="rows"` corresponds to this `:are sheet | rows`, `connect="done"` corresponds to this `:is sheet | done`, and `connect="task-name"` corresponds to this `:is sheet | task-name`. These translations are done automatically by the Quilt engine by inspecting the structure of any spreadsheet trees loaded in the forest, as per Section 6.3

CTS engine would then realign the relations between cells in that row and the nodes within the new `li` element.

2. `Update(TRUE)` would be sent to the cell in the Done column of the new row. This would cause the cell to change its value to `TRUE`.
3. `Update(DefendThesis)` would be sent to the cell in the task-name column of the new row. This would cause the cell to change its value to `Defend Thesis`.

If the user were watch this taking place with high latency, she would see the row in the spreadsheet first be duplicated, then have one value updated, and finally the other.

6.5. User Study

We performed user studies of 15 non-programmers to understand whether Quilt's approach can be learned and applied by individuals with only basic spreadsheet and HTML editing knowledge. Specifically, our study examines three questions: armed with a basic understanding of Quilt, can end users:

1. Connect an HTML interface mockup to a spreadsheet to create a web app?
2. Design and build a spreadsheet to support an HTML interface mockup provided to them, and then connect it?
3. Debug Quilt applications with only a browser and the spreadsheet?

Thirteen of the fifteen subjects were able to complete all tasks given to them. Combined with participant surveys and interviews, the data suggests Quilt's approach takes a step toward enabling this population to construct a class of simple interactive web applications they are otherwise unable to build.

Participants

We recruited 15 participants by visiting four local-area programming education organizations that provide introductory web page authoring classes. According to our survey, six participants had only been using HTML for "a few months" or less,

6. Reactive Programming, Data, and Computation

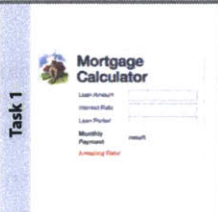
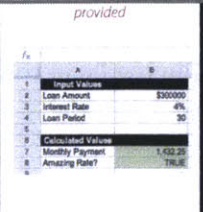
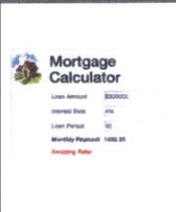

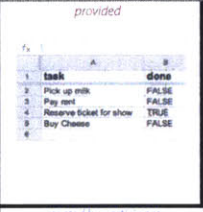

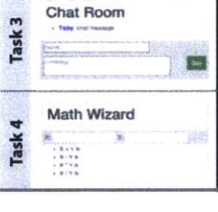
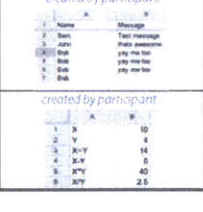
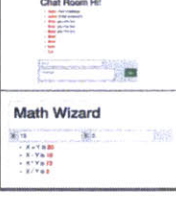
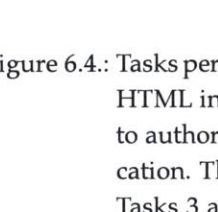
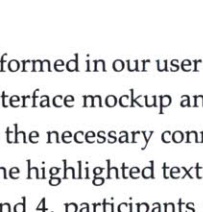
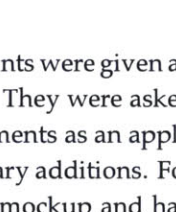
	HTML Interface Mockup <i>provided for all tasks</i>	Spreadsheet for data and computation	Quit Connections participants needed to add the highlighted sections	Finished Web App backed by the spreadsheet
Task 1			<pre> <tr> <td>Loan Amount</td> <td><input connect="B2" /></td> </tr> .. (other two inputs now shown) .. <tr> <td>Monthly Payment</td> <td connect="B7">result</td> </tr> <tr> <td show-if="B8">Amazing Rate!</td> </tr> </pre>	
Task 2			<pre> <ul connect="rows"> <input connect="done" type="checkbox" /> Example Task <form connect="rows"> <input connect="task"/> <button>Add</button> </form> </pre>	
Task 3		<i>created by participant</i> 	Omitted for space. Required: two connect="rows" four connect="column name"	
Task 4		<i>created by participant</i> 	Omitted for space. Required: six connect="(cell coordinate)"	

Figure 6.4.: Tasks performed in our user study. For Tasks 1 and 2, participants were given an HTML interface mockup and a spreadsheet containing data. They were asked to author the necessary connections to connect the two documents as an application. The highlighted text in the HTML contains the necessary additions. For Tasks 3 and 4, participants were only given a user interface mockup and had to additionally design and implement the spreadsheet necessary to implement the mockup.

nine for a year or less, and all but one for two years or less. Nine participants reported that they cannot write HTML without frequently referring to documentation, nine had never used the HTML `form` tag², and five had never used the `table` tag. All participants had used a spreadsheet, and all but one had used Google Spreadsheets. All but four had used a spreadsheet formula. Only four reported that they “understood the concepts” of programming. Of these four, only two had

²That most participants had not used the `form` tag is indicative of how hard it is for this population to build read-write web pages, as `form` elements are only useful in conjunction with a server-side app.

written a computer program before. Both had "about a year" of experience and self-rated as beginners.

Teaching Session

We provided each participant with a fifteen minute private teaching session that was scripted by a worksheet. Using examples, this three-page worksheet presented the main idea behind Quilt (2 minutes), explained the difference between "cell based" thinking—good for fixed inputs and outputs like a math equation—and "table based" thinking—good for lists of items with named properties (3 minutes), and introduced Quilt commands using an example weather report web site backed by a spreadsheet of forecasting data. Participants used this app to practice displaying data, accepting input, and showing/hiding HTML elements for both cell- and table-based addressing (10 minutes).

Tasks

We began the study with only two tasks, but expanded this to four half-way through because participants were completing the tasks with greater success and in less time than we anticipated. We wanted to both take advantage of the extra time and also pro-actively address the concern that our study was not targeted at the right level of difficulty. Eight subjects were only presented the first two tasks, and the final seven were given all four. Each task, as well as screen shots from participant sessions, is shown in Figure 6.4

For each task, we provided the participant with three windows: a web browser with a Google Spreadsheet, a text editor with an HTML page, and a web browser rendering that HTML page. The text editor was pre-loaded with an HTML mockup for an application we wanted them to finish. This allowed us to study the participant's spreadsheet and Quilt activities without worrying about the separate task of web design, since in theory they all knew basic static HTML authoring. This UI mockup was pre-connected to the Spreadsheet (with a `script` and `link` tag in the head) but contained no other Quilt commands. The subject was briefed about the application purpose using a script and then asked to build it.

Tasks were always administered in the same order, and we recorded each task with a screen capture program. The participant was asked to work slowly and speak their thoughts aloud as they programmed.

6. Reactive Programming, Data, and Computation

During each task the experimenter was silent except to notify the participant if they had made an undetected spelling error such as typing `conect` instead of `connect`. In these cases, we waited until they realized themselves there was an error and begun hunting to fix it, and then informed them *“you have a spelling error somewhere in what you’ve typed”*. We did not inform them of the location of the error.

We included these interruptions in the experiments because we did not want our (novice-level) subjects to mistake accidental spelling error for core conceptual errors, thus spawning a wild goose chase of conceptual reassessment when their core understanding was actually correct. All popular programming text editors already have support for exactly this kind of error reporting—and in fact more promptly than we delivered it. But since we did not have a grammar file to describe the CTS language to our text editor so that it could automatically highlight malformed statements, and since the Quilt-variant of CTS syntax is only four keywords, we opted to perform this syntax validation via experimenter interruption.

Tasks 1 & 2 were designed to test whether the subject could learn Quilt and apply it without assistance. In these tasks, the user was provided a finished spreadsheet along with the HTML mockup. The activity required of them was to identify how this mockup should be connected to the spreadsheet to turn it into an application and to make those connections correctly. Task 1 was a mortgage calculator, requiring cell-based connections and the ability to perform output, input, and conditional visibility. Task 2 was a To-Do list, requiring table-based connections and the ability to perform enumeration, output, and row-appending operations.

Tasks 3 & 4 were designed to see if participants were capable of designing and implementing a spreadsheet-based data model to support a Quilt-based application. For these tasks, we provided them with only an HTML mockup and an empty spreadsheet. The activity required of them was to identify the data needs of the UI mockup, construct a spreadsheet to meet those data needs, and then use Quilt to connect the two. Task 3 was a chat website, in which users could post messages to a bulletin board. Task 4 was a math tutor that displayed various arithmetic functions of two input variables.

6.5.1. Result: Learning and Applying Quilt

Before learning Quilt, we showed each participant screen shots of Tasks 1 and 2 and asked how long it would take them to connect the finished web mockup, if

provided, to a data source. Most estimated it would take many hours or workdays. Several provided time estimates but added that they had no idea how to actually go about the task, and two said they would not be able to complete the tasks at all ("not with all the time in the universe" said one). Inexperience likely drives much of the high estimate variance, but these estimates at least indicate perceptions of task difficulty.

After learning Quilt, fourteen of the fifteen participants completed Task 1 without any assistance, and thirteen of fifteen completed Task 2 without any assistance. The time it took to complete these tasks was four minutes or less for most participants, dramatically less than their estimates. Figure 6.5 shows timing information.

While the web apps in these tasks were simple, they required a set of skills these participants largely lacked using traditional methods: reading, writing, and modifying dynamic web data, appending new data items, and conditionally displaying user interface elements. That they could learn to accomplish these tasks with Quilt in only fifteen minutes indicates that Quilt's programming model translates these core data management operations into an approachable interface for non-programmers. One participant had neither written a computer program nor used HTML's `form` or `table` before. Before learning Quilt, she was unable to describe how one might build the Mortgage Calculator or To-Do List with current technologies. After learning Quilt and completing both tasks successfully, she remarked:

It feels amazing. I thought it would take me three 8 hour days to do something like this. And then to be able to just do it with Google Docs which is something that I use so frequently, and for it to be so easy and to make so much sense, yeah, it's really cool.

The tasks were not all completed correctly on the first try, however. Twelve participants experienced a non-spelling error in at least one task, and five experienced an error in both tasks. In all cases where the participant ultimately completed the task correctly, they were able to identify and fix the error by reloading and sometimes experimenting with the web page and then altering the code they had written. More about this debugging behavior is in the Debugging section.

6. Reactive Programming, Data, and Computation

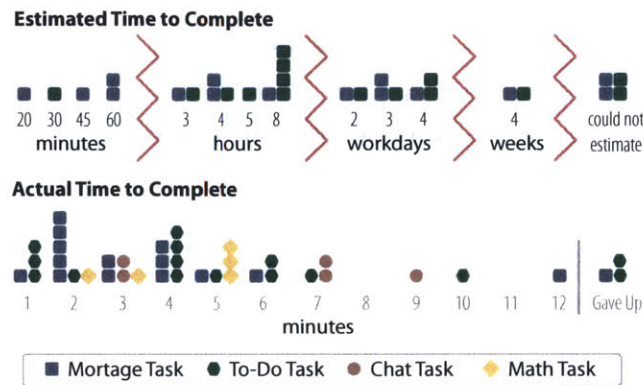


Figure 6.5.: Estimated and Actual time to complete tasks in our study. Please take careful note of the x-axis units: most participants estimated that it would take them many hours or days to complete these tasks before learning Quilt. The actual time to complete these applications was a few minutes.

6.5.2. Designing Spreadsheets to Back Web Apps

Recall that Tasks 3 and 4 were added half way through the user study to see what would happen when participants were burdened with more of the work. These tasks, shown in Figure 6.4, only provided a web interface mockup. Participants had to create a spreadsheet to support that mockup themselves and then connect it to the HTML page.

We intended to offer seven participants Tasks 3 and 4, but two had to leave after completing Tasks 1 and 2 and could not be offered the tasks. Of the five that remained and were offered the tasks, all completed both tasks successfully. Their times are shown in Figure 6.5.

How difficult was this additional data modeling step? While we can not make strong conclusions from only five data points, timing information from the videos along with interviews was interesting. All five participants spent less than one-third of time working inside the spreadsheet for the Chat application---a table-based app. But four of five participants spent more than half of the time working inside the spreadsheet for the Math application---a cell based app.

Some of this difference is simply due to the fact that the math application required more spreadsheet work. But the recorded videos and audio make it clear

that the lack of spatial coherence when addressing cells individually posed problems as well. All five users wrote the Chat spreadsheet (a table) correctly on their first attempt, but only two wrote a working Math Tutor spreadsheet without having to make structural changes. The common problem seems to have been that the HTML mockup contained a *list* of math functions, but the data model to support this list did not have any particular list structure to it. This left them unsure of the physical arrangement of data in the spreadsheet to support this interface.

One caught her mistake almost immediately, but the other two created finished tables of math functions, repeating placeholders for the X and Y variables once per row, with a different math operation in each row. They caught their errors when trying to connect the spreadsheet to the web mockup. At this point, the modeling error was apparent because it was not clear how to make the connection. The web site only had one X and Y input, but their spreadsheets had four cells reserved for X and four cells reserved for Y. In fact, only a single cell for each of X and Y was needed, and all the math equations referenced this same X and Y. When asked what he was thinking the moment he revised the table, one replied, "Once I went through the process, used my hands instead of my brain, it started making more sense. Once I started trying to connect it to the HTML."

6.5.3. Debugging Behavior

Quilt does not offer any particular debugging assistance past that already in the browser and spreadsheet. One important question is how to best service the debugging needs of SV app authors who are unfamiliar with traditional programming and debugging concepts. To make progress toward answering this question, we coded all *successful* Task 1 and 2 videos to mark when non-spelling errors occurred and how long it took to fix them after being noticed. In these 27 videos, we found 19 errors across 12 (of 15) participants, all of which were identified and fixed by the participant without assistance. The time it took to fix these errors had high variance—the standard deviation (77s) was above the mean (69s)—likely due to different skill levels and error context.

While the data set is too small to make strong conclusions, the errors could be clustered into three categories. **Static errors** (13 recorded) were those immediately apparent upon viewing the web page, such as connecting and overwriting the wrong HTML element. No matter what the spreadsheet value, these are obvious. **Dy-**

6. *Reactive Programming, Data, and Computation*

namic errors (3 recorded) were those that might be immediately apparent, but might also require experimenting with different data inputs to surface, such as connecting a `show-if` to the wrong cell. If the right cell and the wrong cell had the same "truthiness" value, the error may not be apparent until data changes to reveal the inconsistency, tempting a novice to prematurely declare a correct implementation. **Omissive errors** (3 recorded) were those due to missing functionality or Quilt statements so mistyped that the page simply did not work but no visible clues were available to help diagnose, such as forgetting to connect the form element. These errors could be as easily fixed by "fiddling" with the HTML markup.

Figure 6.6 shows the most elaborate static error encountered: nearly the entire Mortgage web page was connected incorrectly, and the user went through a sequence of iterative modifications until figuring out the proper HTML nodes and spreadsheet cells to connect. At each step in Figure 6.6, the existence of the error was so clear that a nearest-neighbor walk toward correctness was possible. This participant recalled his thought process afterward, "I tested it out and it changed to the wrong thing so I'm like, 'OK. That's not right.'" Another participant who made an identical error said aloud while programming, "Alright so I put the connect in the wrong place. It needs to be connected to the input area."

While these categories may be useful in framing future work, other groupings may prove more appropriate with more data, e.g. connection, conditional, and form errors.

6.5.4. Overall Reactions

Overall reactions to our method were enthusiastic, as the population we studied appeared to be right in the middle of a demographic capable of quickly learning Quilt but not capable of building Quilt-style applications otherwise. Participants liked the immediate feedback and many saw it as a prototyping tool because of that. One participant cited the lack of needing to understand how to write sequential steps often required by programming as a benefit: "For somebody who is very familiar with spreadsheets it makes sense in my head. It doesn't require a lot of 'if this happens, this happens, if this happens this doesn't happen.' It just naturally makes sense,"

All reported that they would use a system like this if it was available to them. "As a beginning programmer, it feels incredible. I feel so much more empowered,"

6.5. User Study

Screen Capture from Editor Window (red boxes added)	Screen Capture from Web Browser
<pre> <tr> <td connect="B1">Loan Amount</td> <td><input /></td> </tr> <tr> <td connect="B3">Interest Rate</td> <td><input /></td> </tr> </pre>	<p>(a) Several errors exist initially: spreadsheet cells have been connected to the web page's labels, rather than input elements, and the first connection was made to an empty cell coordinate (B1).</p>
<pre> <tr> <td>Loan Amount</td> <td connect="B1"><input /></td> </tr> <tr> <td>Interest Rate</td> <td connect="B3"><input /></td> </tr> </pre>	<p>(b) Seeing the rendered web page in the prior step, the participant realizes he has made connections to the wrong nodes. He copies and pastes the connect statements to the other <TD> elements, causing the cell values to overwrite the <INPUT> elements.</p>
<pre> <tr> <td>Loan Amount</td> <td><input connect="B1" /></td> </tr> <tr> <td>Interest Rate</td> <td><input connect="B3" /></td> </tr> </pre>	<p>(c) The participant exclaims "Oh" and makes a reference to the fact that he was skimming the HTML without taking time to read it, then moves the connect statements into the <INPUT> elements.</p>
<pre> <tr> <td>Loan Amount</td> <td><input connect="B2" /></td> </tr> <tr> <td>Interest Rate</td> <td><input connect="B3" /></td> </tr> </pre>	<p>(d) When one of the input boxes fails to load a value in (c), the participant notices the null value problem for the first time. After re-checking the HTML attributes, he notices and fixes the cell reference (B1 -> B2) after re-examining the spreadsheet.</p>

Figure 6.6.: Most errors encountered could be identified and diagnosed by simply reloading the web page. This participant had the longest sequences of such activity. Other errors were harder to detect, requiring the user to experiment with different data values or manually inspect code to diagnose.

one said. And another: "This is great because it lowers barrier to entry for someone who is never going to be a developer."

Several users asked to follow up and use Quilt in their projects, which varied as widely. Many wanted to add a simple form to their personal or company site, but control the styling of the web site in a way that form software (like Google Forms) does not allow. One wanted to create a tool to help track her meditation practice. Another wanted to quickly parameterize web designs her company sells with customer information. And one wanted to use Quilt to test different landing pages for his plumbing company by having multiple forms that all fed to the same spreadsheet. We are following up with these users and others in a future iteration of the prototype.

6.6. Discussion

6.6.1. Low level end-user programming

Quilt's approach is quite low-level compared to related work. Spreadsheet-backed visualizations using the Exhibit framework [48], for example, are built by invoking pre-made widgets from HTML tags. Quilt offers only the ability to connect raw HTML elements to spreadsheet entities. This puts the non-programming population in a position where they come very close to needing to think like programmers.

Concepts like the `show-if` command require understanding a bit about boolean logic. `show-if` and `hide-if` were kind of confusing. `show-if`, show if what? It would make more sense if you called it `show if true`," said one user. Others made mistakes concerning the notion of pointers, such as typing `connect="Loan Amount"` instead of `connect="B2"`. This suggests a more flexible addressing scheme that allows users to reference values by nearby cell labels may be useful.

There was also a physicality to some participant's understanding of the task at hand. Three users reported becoming confused when they realized they would need to connect a vertically organized UI element (e.g., Todo list) on the web page to the keyword `rows`, which they thought of as something horizontal. Despite these confusions, the fact that nearly all participants identified and corrected their bugs suggests that the web relations approach reduces the search space of possible programming statements to a set navigable by authors with only partial understanding of the concepts involved.

Participants did report having to adjust expectations to get used to the idea that one relation (e.g., `connect`) could provide so many different behaviors (output, append, modify, etc). One remarked, "I'm doing something totally different but it's the same wording," and this ran counter to her understanding of programming having many commands. Another commented "I have a sneaking suspicion this won't work," while typing `connect` into an input element after having just used it on a `span`. Ultimately, participants seemed to like this approach after getting used to it. "I felt like there were very few functions I needed to remember," one said when asked what she liked about Quilt, and another: "[I felt] overwhelmed, but then I knew I need to figure out where I will put my connects." The relational approach to templating offers the possibility that, in many cases, knowledge that two data objects should be "connected" is sufficient, without understanding the details

of how to implement this connection.

In some cases, Quilt's compact programming language *induced errors* because participants got so used to typing `connect` they forgot other commands existed. One participant used `connect` for everything in Task 1, before realizing one of the elements required a `show-if`. She remarked, "I think it was just a matter of momentum. I was just connecting everything. It just seemed like `connect` would just work because everything else just worked with `connect`." Another who made the same mistake said, "I didn't think it through. It was just a natural reaction, because I was connecting a lot of things." One caught himself making this error and muttered, "It ends up being very repetitive. I must not get cocky."

6.6.2. Design Patterns

Because Spreadsheets are Turing complete, the spreadsheet-view model can support arbitrarily complex application logic in theory. But the kinds of applications for which SV programming is a *practical approach* for end users will likely be impacted by design patterns developed to handle common scenarios. This is an interesting and fertile ground for future work. We provide two examples here.

Functional View Pattern

It is clear that information management applications have data models. But they also have view models, which represent a transformation of the data model that corresponds to what should be shown in the UI at any given time. For example, the state of a sortable table's sort configuration, a phrase typed into a search box, a checkbox state, or some choice made on a previous page (such as the "Safe Mode" option on many search engines). In web applications, the view model is often left implicit, scattered across HTML, Javascript, and server-side state. This obfuscates a clear definition of view state in a way that complicates development and debugging.

SV applications provide an attractive way to make these concepts isolated, explicit, and functionally dependent by defining them as separate spreadsheets. Figure 6.7 shows this pattern, which is similar to the organization we used in the wedding planning app in Figure 7.2. Using inter-document formulas, the view model sheet is defined as a set of spreadsheet formulas that compute upon the data model sheet. These computations are parameterized by data in the parameters sheet.

6. Reactive Programming, Data, and Computation

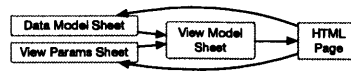


Figure 6.7.: The Functional View Pattern, in which data for the HTML view is stored in a view model sheet. This sheet is a function of a data model sheet and view parameter sheet.

In this pattern, the HTML page displays data from the view model, but it writes data back into the data model and view parameter sheets. Google Spreadsheets natively causes these writes to propagate to changes in the view model, and Quilt causes changes in the view model to propagate forward into the web page. Quilt's spreadsheet addressing language is easily extended to specify which sheet, among many linked sheets on a page, is being referenced.

Quilt's current implementation supports this pattern, but exporting all view logic to a spreadsheet introduces UI latency. This is a problem of system implementation though, not one of end-user programming model design: related work [9] has shown it is possible to automatically rewrite portions a web app's data and computation and relocate it in the web browser for performance gain. The spreadsheet could thus serve as a useful authoring and debugging environment even if it is not the execution platform.

Introducing Multiple Users

Quilt supports multiple simultaneous users with a modification to the functional view pattern shown in Figure 6.8. In this pattern, spreadsheets can be designated as either *global* or *per-user* in the `link` element that identifies them in the web page's head. When a new user visits the web app, Quilt makes fresh duplicates of any sheets marked as *per-user*, creating a sandboxed data store for that user that can remain persistent through multiple visits. The *per-user* sheet may include both view parameters and user data, and the shared data sheet may aggregate values from many separate *per-user* user sheets. This pattern also enables users to store data that is private to them, and avoids a problem in which simultaneous users might overwrite each other's view parameters (if they were sharing a global view parameter sheet).

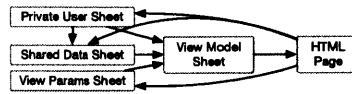


Figure 6.8.: Modification of the Functional View Pattern to support multiple simultaneous users and private data.

6.6.3. Copy-Paste-Modify

Like Exhibit and data-oriented projects such as ReForm [87], Quilt supports a model of authoring that should lend itself well to copy, paste, and modification by others. Interestingly, "others" in this case may not just be other application creators but also the users of an application, people who just want to get some job done.

Most software today is opaque from the standpoint of the user. Only programmers using open-source software have the ability to modify and recompile it for themselves. Quilt applications, being just regular HTML plus Spreadsheets, might be seen as an functional analogue of open-source applications for end-users.

A user of a todo list can easily add an extra Due Date field (one `` in the web page and one column in the spreadsheet). A mortgage calculator user might add extra fields to represent other utility costs, transforming it into a cost of living calculator (a few HTML nodes, a few spreadsheet cells, and a modified equation).

Even if a user did not care to modify their application, they could at least open up the spreadsheet and have a reasonable idea of underlying data that was being tracked to provide them with the application's user experience. They could send it to a friend, graph it, fix bad data, or export it for use with another tool.

6.6.4. Visual Programming

Many WYSIWYG environments for HTML authoring already exist, and spreadsheets themselves are a visual programming environment. By grounding Quilt in an HTML-based syntax and existing spreadsheet software, rather than creating a custom HTML+spreadsheet WYSIWYG application, we ensure that this method works independently of any WYSIWYG tools that may be layered on top. From an authoring UI perspective, Quilt only introduces the new step of crafting declarative relations between HTML elements and spreadsheet elements. Quilt could thus easily be made a visual programming environment by making this relation crafting

6. Reactive Programming, Data, and Computation

step visual.

Such an environment might present the author with a split-screen view of their spreadsheet and their HTML UI. The author would then visually drag and drop spreadsheet entities (cells, rows, columns) onto the web page (or vice versa) to draw relations. After drawing a relation, the user selects what kind it is (connect, delete, show-if, or hide-if). Because web relations annotate ordinary web pages, this visual development environment could enable the user to interact with and debug the live page while editing it.

Web Worksheets embedded in Spreadsheets Quilt demonstrates the potential of offering more flexibly designed spreadsheet UIs inside the spreadsheet application itself. Large spreadsheets with many worksheets for data and computation could include “web worksheets” written as HTML to provide a better user interface to the analyses than a tabular view provides. Applications built this way could contain rich UIs while relying upon a well-understood infrastructure that additionally permits users to view and edit the data and computation layers.

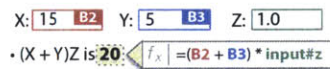


Figure 6.9.: Mockup of what spreadsheet-style editing directly in a web page might look like, enabling spreadsheet elements (B2 and B3) to be mixed with HTML elements (input#z).

These packaged web worksheets could even be editable in spreadsheet fashion, like the mockup in Figure 6.9. A user could select a web element, revealing a formula entry widget to bind the web element to a formula result. A user might even mix spreadsheet cell addresses with CSS selector addresses. The total amount for a restaurant bill might be the sum of column B, plus the Tip entered in a web input box, for example: `=sum(B:B, input#tip)`. A basic example would be re-implementing the spreadsheet UI as a table element, with each table cell representing a spreadsheet cell. In many ways this would be similar to a read-write, page-wide variant of WebCharts, which is a framework for creating web-embeddable visualizations that display data in spreadsheets [32].

6.6.5. Future Work

In addition to performance and design pattern questions raised by our implementation, there are many intriguing questions to explore surrounding spreadsheet-view applications more generally. Understanding what class of applications can practically be authored using this approach, and the kind of user scaffolding required to create nontrivial applications is a key next step. Quilt's choice to use HTML and Spreadsheet software *as-is* enables it to work together with the many tools that already exist to support authoring in each respective medium. But there are also many benefits to be explored by taking a joint approach: modifying the spreadsheet software to embed awareness of the connected web page and explicit UI support for SV application authoring. Chang's work with Gneiss, done in parallel to our work on Quilt, explores this approach [21].

7. Discussion

The preceding chapters have probably spawned many questions and hopefully a few good ideas, too. This chapter offers a discussion of some of the higher-level concepts, limitations, and possibilities surrounding this work.

7.1. Can this Relational Model Build Non-Trivial Applications?

One reasonable question is whether a language like CTS can scale to the project complexity of real-world applications like Facebook or Amazon. An underlying cause for this concern is the seeming asceticism of the relational approach CTS takes. Real world applications are filled with special cases, last minute data manipulation, and odd situations. Our template languages tend to address these special cases by allowing procedural code as well as templating directives—essentially an `eval()` escape hatch. Can a handful of relations and tree manipulations truly do the job our template languages provide?

Given that no large-scale applications have yet been attempted using this approach, a one-word answer to this question is not possible. But the following points suggest that the answer is *yes*:

Supplementation, not Replacement. Web programming by definition is a hybrid blend of several languages at once, and using any one does not preclude use of another. CTS offers improved HTML editing and management (the template problem) and a language for specifying how data should synchronize between objects (the reactive programming problem). But there are many operations that CTS does not do. For example, creating a drag-and-drop user interface requires event-based programming and pixel-based addressing. But other languages (or libraries within Javascript) do not demand exclusivity over the web page in the same way

7. Discussion

that CTS does not demand exclusivity. The trick is to find the right division of concerns so that the blend of frameworks coexisting on a web page can work together effectively.

Perhaps Computation Doesn't Belong in Templates. One contrast between CTS as a template engine and typical template engines is that CTS does not support any data computation. Instead, it requires the data computation to be performed by some other system (e.g., a spreadsheet or server-side code) that exposes the results of the computation for CTS to use as a tree. This means that a string joining operation such as `{{firstName + ' ' + lastName}}` is impossible with CTS. While some may see this as a concerning design choice, we push back with the proposal that computation might be better isolated and defined outside the view layer of an application. This preserves the view as a pure design mockup and forces developers to be explicit about the kind of data they intend to deliver to that mockup.

Filtering and Data Formats are Easy Extensions. While CTS does not support computation as part of its template rendering, there are some operations such as filtering and data formatting that are trivial to add.

Some of these filters use a small extension to CTS not discussed for the sake of simplicity. In many languages, it is useful to be able to *reify* a statement (e.g., to say something like, "That statement is true"). Reification in CTS refers to the fact that the CTS author is able to make key-valued statements commenting about relations. Specifically, three optional key-value objects can be attached to each relation: one at each binding point and one on the edge. For example, perhaps the tree to which n_2 belongs is read-only. A relation might encode this as a reifying statement `readonly: true` on the binding point with n_2 .

Figure 7.1 illustrates a relation in CTS. The relation $r(n_1, n_2)$ is shown in black and reifying statements about that relation are shown in red.

When writing CTS, these objects can simply be placed as JSON dictionaries after each component of the CTS statement: the two selectors and the relation in the middle.

Here are a few filters that the CTS runtime engine supports, but were not discussed in this thesis, just to get a feel for them. Some are invoked via reification:

- **Number Formatting.** Displaying an integer as a currency, for example, can

7.1. Can this Relational Model Build Non-Trivial Applications?

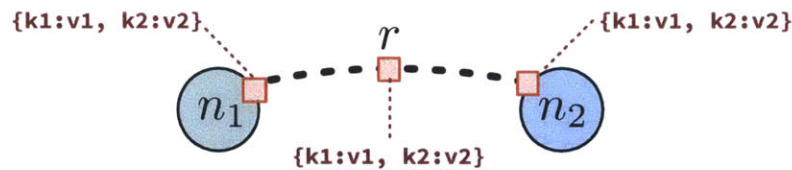


Figure 7.1.: Relations in CTS are binary and bidirectional. They exist between two tree nodes. These nodes can be from the same tree or from different trees. Each component of the relation (the relator, or either node's participation) can be reified by a key-valued property list.

easily be tacked on as a reification to a relation. For example, consider the following relation which binds a span object to a spreadsheet cell:

```
sheet | A1 :connect span {type: float, precision: 2}
```

Such a reification could make sure that any data traversing in or out of that span element should be first cast as a floating point number and then rounded to a precision of two decimal places.

- **Prefixes and Suffixes.** Some text filters, such as a prefix or suffix, are also easy to implement as bidirectional filters. Consider the following statement:

```
json | .name :is h1 {prefix: "Name: "}
```

This will ensure that any data flowing into the `h1` will gain the prefix `Name:` , and any data flowing out of `h1` will be stripped of that prefix.

- **Striped Enumerations.** Sometimes it is useful to have every i^{th} element of an enumerated set take on a different design appearance that may not be achievable with CSS alone. When the CTS engine is cloning set children to add to the out-degree of an `are`-connected node, it takes the initial set of children of length i and then clones $j \bmod i$ for every future j^{th} child. This enables a design mockup to provide an enumerated list that cycles through a set of different designs.

The broader point is not these specific examples, but rather to show that there are certain classes of data filtering that lend themselves well to the relational approach without breaking the lossless rendering properties of the CTS engine. A production

7. Discussion

version of CTS would likely enable extensions of this type that could be invoked via the three reification dictionaries reserved for each relation.

Modern Spreadsheets Add Database-like Power Web spreadsheet software (like Google Spreadsheets) also supports many database-like operations not supported by desktop spreadsheets. Formula-based filtering, sorting, and table-joining, as well as inter-document addressing and web API calls are all possible and can be parameterized by other cells. This opens up real possibility for spreadsheets to provide a programming environment for non-trivial web applications. Figure 7.2 shows a portion of a multi-sheet wedding planning app we built with Quilt to demonstrate this functionality. When the web user types into a search box on the web page, Quilt synchronizes that with a cell in a **Parameters** worksheet. This causes a column in the **RSVPs** sheet to recompute whether each row is a search result. Quilt synchronizes the new column data to the browser, where it toggles visibility of each list item.

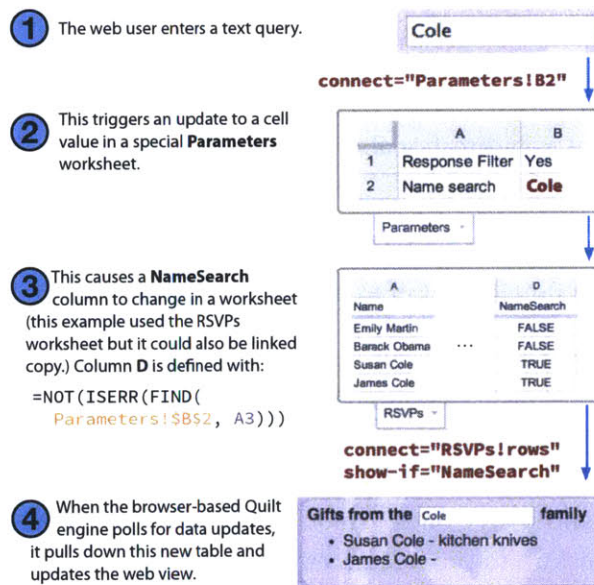


Figure 7.2.: Demonstration of how Quilt's two-way data syncing enables data operations like reactive text search using web elements to parameterize spreadsheet functions.

7.2. Are Other Relation Sets Viable?

The particular relation set of `is`, `are`, `graft`, `if-exist`, and `if-nexist` was chosen because it represents core functionalities of template languages. And the semantics of those relations with regard to the CTS rendering engine are carefully designed to guarantee the preservation of certain information in the output.

That said, there are clearly different sets of relations that might be equivalent or better in practice than the one proposed. The Quilt system was able to collapse `is`, `are`, and `graft` into a single relation `connect`, for example, because of the additional type information available when binding to a spreadsheet tree.

Even in the standard set of relations, `is` and `graft` are extremely similar in construction. The only difference is that `is` transcludes a block of content without processing it for further CTS relations, whereas `graft` transcludes a block of content that is assumed to contain further CTS relations. One could collapse the two into a single relation, but we decided to keep them separate for look-and-feel reasons: the idea was that `is` feels like a print statement and `graft` feels like a subroutine, which is a natural way for programmers to think.

7.3. Does the Schema Alignment Problem make Post Hoc Retargeting just an Academic Exercise?

First of all, the schema alignment problem will always exist (modulo a philosophical debate). Whenever many people generate individual expressions of the same coarse domain concept in isolation, eventually somebody will generate an expression that is structured differently than the others. From this perspective, the presence of CTS or any other retargeting system does not create the problem, but merely surfaces it to the user.

Practically, in the case of CTS, here is how someone seeking to retarget a page would handle the problem:

- When a design styles content that you don't have, simply create an `if-exist` relation between that region of design and some selector on your content page that matches the null set of nodes. When the final page renders, that design element will not appear.

7. Discussion

- When you have content that a design mockup doesn't style, this is a bigger problem. You could find an unused portion of that page and use an `is` relation to shove your content in there, but that might not always be a realistic option.

For this latter case, and even in the former, manually copying the design and then modifying it may be necessary.

There are plenty of cases that would work, however, depending on the two pages involved. As general heuristic based on retargeting exercises performed as part of the CTS work, display semantics (title, sidebar, main body, footer) tend to cause fewer alignment problems than fine-grained data semantics (middle name, area code, sales tax).

If communities of CTS microformat authors did, in fact, emerge, the existence of common microformats should encourage some degree of schema convergence within the community.

8. Conclusion

Before the era of relational databases, data management applications still existed--they just were not unified under a single data model. Techniques for constructing these applications were varied and customized to each new project. Retrieving information, storing information, and combining information depended on programming choices each developer made. As a result, building these applications was costly and the data, being tied so tightly to the application code, was hard to reuse. Relational databases revolutionized how we manage data by providing a data model, a calculus for operating on that data model, and the ability to encapsulate data concerns away from application concerns.

Many elements of web authoring today look like data management applications before the relational database was introduced. Our web authoring practices tend to approach the task as one of programming, and thus the web apps we create are long procedural programs. Synchronizing information between client and server, reusing designs, or combining different design elements and content elements tend to be performed by bespoke application code, making them hard and expensive to reuse. We lack a model to describe the way pieces of a web application fit together, a calculus for operating on that model, and a way to encapsulate all the pieces that are ultimately woven into a web site.

This thesis developed ideas for improving web authoring practice by adding a relational layer to web content to bind structures on the web to other structures on the web. Based on this core model, it introduced a carefully designed set of relations that act like keywords in a programming language, enabling web authors to stitch together content, reuse designs *post facto*, and construct reactive information management applications, all just by writing a few relational statements.

The resulting language, called Cascading Tree Sheets, is presented to the author with a surface form aimed at usability by novices and experts alike. It borrows the notion of selectors and microformats from CSS, enabling groups of authors to converge on a common set of classes to describe and invoke certain functionality.

8. Conclusion

Once someone creates a tree sheet, others can use it simply by applying the right CSS classes to their HTML page. And the Quilt application engine leverages our familiarity with spreadsheets to enable interactive read-write-compute web pages without having to author Javascript or server-side code.

Cascading Tree Sheets enables is better organization and encapsulation of web content, and studies have shown that this improves authoring efficiency and enables authors to build web applications using UI mockups rather than fragmented template modules. CTS also preserves enough information about the structure of a page that a runtime engine can reason about how changes in one place should ripple to another—the core feature of reactive web programming. The Quilt engine showed how combining this capability with spreadsheets enables easy construction of simple information management applications, and more broadly this ability opens the door to better user-driven content management.

The web is place of thriving platform experimentation and standards building. To those participating in both of those activities, this work is a stake in the ground toward the goal of improving the foundational languages that underpin the web so that we might simplify and improve the application building layered on top.

Appendices

A. CTS Grammar

This section provides a formalization of the grammar used to author a tree sheet in Extended Backus-Naur Form. EBNF [78] introduces additional operators over BNF that permit a more concise grammar description. Relevant to the CTS grammar is EBNF's use of square brackets $[]$ to denote an optionality, curly braces $\{\}$ to permit repetition, and parenthesis $()$ to denote grouping. For example, the EBNF statement $\{A\}$ corresponds to the language recognized by the regular expression A^+ , the EBNF statement $[\{A\}]$ is the same language recognized by the regular expression A^* , and $[\{A\}](b|c)$ is the language recognized by the regular expression $A^* [bc]$.

Tree Sheets A tree sheet is a collection of CTS statements. Every tree sheet has two sections, a header and a body, both optional.

$$\langle TreeSheet \rangle \rightarrow [\langle Header \rangle] [\langle Body \rangle]$$

The Header The header contains a set of *At-Rules*, modeled after the same statement type in CSS. CTS uses two kinds of At-Rules: tree rules instantiate and name trees, and resource rules instantiate resources (like CSS or Javascript files) that should be included as a supplement to those trees. Valid tree types are HTML, JSON, and Google Spreadsheets, and valid resource types are CSS, Javascript, and CTS.

Optional arguments may be provided to either type of rule. They consist of zero or more strings followed by an optional JSON-style dictionary that acts as a grammatical outlet any arbitrarily complex configuration that may be necessary.

$$\langle Header \rangle \rightarrow \{ \langle AtRule \rangle \}$$
$$\langle AtRule \rangle \rightarrow \langle TreeRule \rangle \mid \langle ResourceRule \rangle$$

8. Conclusion

$\langle TreeRule \rangle$	\rightarrow ``@" $\langle TreeType \rangle$ $\langle TreeName \rangle$ $\langle URL \rangle$ $\langle Arguments \rangle$ ``;"
$\langle ResourceRule \rangle$	\rightarrow ``@" $\langle ResourceType \rangle$ $\langle URL \rangle$ $\langle Arguments \rangle$ ``;"
$\langle TreeType \rangle$	\rightarrow ``html" ``json" ``gsheet"
$\langle ResourceType \rangle$	\rightarrow ``js" ``css" ``cts"
$\langle TreeName \rangle$	\rightarrow $\langle SimpleString \rangle$
$\langle Arguments \rangle$	\rightarrow [$\langle SimpleString \rangle$] [$\langle DictArg \rangle$]
$\langle URL \rangle$	\rightarrow A string as defined by the grammar in RFC 1738 [13].
$\langle SimpleString \rangle$	\rightarrow A string matching the regular expression [A-Za-z0-9_-]*
$\langle DictArg \rangle$	\rightarrow A JSON-style dictionary as defined by the Javascript Object Notation grammar in RFC 4627 [27].

Here are several examples to illustrate the grammar governing the head of a tree sheet.

- The web page at `edwardbenson.com`, to be called `ted`:

```
@html ted http://edwardbenson.com ;
```

- The worksheet `Sheet3` of the Google Spreadsheet at URL `$URL`, to be called `inventory`. The URL is denoted as `$URL` simply because Google Spreadsheet URLs are so long they distract from the explanation on a page of text. The worksheet identifier `Sheet3` is provided as a string argument.

```
@gsheet inventory $URL Sheet3 ;
```

- That same Google Spreadsheet with additional attributes stating that read-only access to the sheet should be attempted without authentication, but write access will require authentication.

```
@gsheet inventory $URL Sheet3 {read: "public", write: "private"} ;
```

- A CSS file to be included in the forrest as an accompanying resource:

```
@css http://example.org/style.css ;
```

The Body The body of a treesheet consists of a set of binary relations between nodes from named trees in the forest. Each of these relations has three parts: two *selectors* and a *relator*. The *relator* is label which specifies the relation type along with an optional JSON-style dictionary. Each selector "selects" zero or more nodes in a tree, also with an optional JSON style dictionary.

$\langle \text{Body} \rangle \rightarrow \{ \langle \text{RelationSpec} \rangle \}$
 $\langle \text{RelationSpec} \rangle \rightarrow \langle \text{Selector} \rangle \langle \text{Relator} \rangle \langle \text{Selector} \rangle \text{ ``;"}$
 $\langle \text{Selector} \rangle \rightarrow [\langle \text{TreeName} \rangle \text{ ``|"} \langle \text{SelectorString} \rangle [\text{DictArg}]$
 $\langle \text{Relator} \rangle \rightarrow \text{ ``:"} \langle \text{RelatorString} \rangle [\text{DictArg}]$
 $\langle \text{SelectorString} \rangle \rightarrow \text{Any string not containing a space followed by a colon.}$
 $\langle \text{RelatorString} \rangle \rightarrow \langle \text{SimpleString} \rangle$

The tree name is an optional prefix for selectors. If no tree name is provided, CTS assigns that selector to the HTML page loaded in the web browser and interprets the selectors as a CSS selector into HTML.

Because the CTS language is intended to make statements about any tree-shaped structure, the grammar for selectors is left slightly constrained but otherwise undefined. The only prohibition on these selector strings is that they do not contain colon-prefixed words (a colon prefixed word is the indication that the first selector has ended and the relator has begun). This is a rather arbitrary rule chosen for easy compatibility with existing CSS, JSON, and Spreadsheet selectors languages. Changing this rule to something more extensible would have no meaningful impact on CTS.

Each selector string is interpreted as a query into the associated tree:

1. HTML trees utilize the CSS selection language [63]. For example the string `div.title` selects all nodes of type DIV that contain the class title.
2. JSON trees utilize the standard JSON object dot-path notation with an optional wildcard to address all elements of an array at once. For example, the selector `blog.posts[0].title` selects the title attribute of the first object of an array named posts in the object named blog. Using a wildcard, the selector `blog.posts[*].title` selects the title of all post object.

8. Conclusion

3. Google Spreadsheet trees are addressed with a special language created for demonstration purposes:

- a) Cells can be selected by typical letter-number strings, such as A3 or Z26.
- b) All rows can be addressed by the keyword `rows`. A particular row can be addressed by the keyword `row(rownum)`. And a named row may be addressed by the keyword `row(.rowname)`, where `rowname` is replaced by the value in the first column of the sheet, naming the row.
- c) Columns can be addressed similarly, using the `cols` and `col` keywords.

For example, the selector `col(.FirstName)` selects cells 2 to N of the column containing `FirstName` in row 1.

B. Formal Rendering Engine Description

CTS relations are simply declarative statements about structure. But we make these relations so that they can be useful to us somehow. This section describes the CTS rendering engine, which allows a forest of trees that have been related with CTS relations to produce a web UI.

At the highest level, the rendering process takes as input the entire forest of trees and relations, along with a designation of a *root* node. This *root* node is a node among the forest where the rendering will begin. It produces a new tree rooted in a clone of *root* but with potentially different structure reflecting the relations that the rendering engine encountered along the way.

$$\text{render}(\text{root}, t_1, \dots, t_n, r_1, \dots, r_m) : t_{n+1}$$

Or simply:

$$\text{render}(\text{root}, \text{trees}, \text{forest}) : t_{n+1}$$

This section formally describes this process as a series of relation-aware tree transducers. We then show that this process is lossless with respect to the *root* tree and relations that participated in decision making.

B.1. Preliminaries

First we begin with some preliminaries to help us develop a symbolic way to describe the types of tree manipulations that CTS supports.

A tree-walking automata is a nondeterministic finite state machine that traverses a tree structure rather than a sequence of input characters [30] [35]. At each node, it can advance into one of the child nodes or to the parent node, as well as change state. The transition function maps the current state state, current node label, and current node index (0 for tree root or i for the i^{th} child of a parent) into a new state and which new node to step into.

Unranked tree automata relax the requirement that a node's label determines its rank according to some rank function $rk : \Sigma \rightarrow \mathbf{N}$. An HTML document is a good example of an unranked tree: an unordered list (ul) node may have arbitrarily many list item nodes (li) as children. There is thus no one particular rank for a node labeled ul. The sequence of children for a node in an unranked tree is called a *hedge* [26].

Tree transducers are a tree automata that additionally output a tree while walking it, just as finite state transducers are to finite state automata¹.

A top down tree transducer is a four-tuple (Q, Σ, Q_0, R) where Q is a finite set of function names, Σ is a finite set of input and output symbols, Q_0 is a set of initial functions, and R is a set of rules of two forms $q(a(x_1 \dots x_n)) \rightarrow t$ and $q(x_0) \rightarrow t$, where q is a function, $a \in \text{Sigma}$, x_i are variables representing trees, and t is a tree.

For example, the following set of rules would copy a tree, replacing all DIV elements with SPAN elements in an HTML tree.

$$\begin{aligned} q(\text{DIV}(x_1 \dots x_n)) &\rightarrow \text{SPAN}(q(x_1) \dots q(x_n)) \\ q(x_1) &\rightarrow x_1 \end{aligned}$$

B.2. A Notation for Unranked Relation-Aware Transducers

The CTS Engine uses a modified *unranked relation-aware top-down tree transducer*. This section develops both the syntax and the concepts we will need to adapt a standard top-down tree transducer into an unranked relation-aware one.

¹FMTs and FSAs instead accepting linear sequences of characters as their inputs instead of tree-structured objects

8. Conclusion

Adaption for Unranked Trees

The basic syntax used for top-down tree transduction does not lend itself well for environments in which the parameterization at run-time is potentially very large. It would not be productive to exhaustively enumerate, for example, the same basic rule for all possible HTML nodes or attributes. So here we extend the syntax to:

1. Provide additional macros to help generalize rule application
2. Incorporate information from the relations on multiple trees as well.

First, we allow for the notion of variables, using a question mark syntax. For example, the variable $?n$ could match any node. A variable's scope is only within the a single rule. Using variables, we might compactly describe the identity tree transducer for binary trees with the following rules, for example:

$$\begin{aligned} \mathbf{Ident}(?n(?l, ?r)) &\rightarrow ?n(\mathbf{Ident}(?l), \mathbf{Ident}(?r)) \\ \mathbf{Ident}(?n) &\rightarrow ?n \end{aligned}$$

Binary trees are useful constructions for their regular structure, and many transduction applications assume them (since all trees can be converted to a binary representation), but we introduce additional macros to help us describe trees of arbitrary arity.

First, we add the rule that a state applied to a list of nodes is the same as that state applied to each node individually. Or,

$$\mathbf{State}(?n_1, ?n_2, \dots ?n_n) == \mathbf{State}(?n_1), \mathbf{State}(?n_2), \dots \mathbf{State}(?n_n)$$

This is a natural extension of the representation considering that these functions are used to represent states. The above two statements both merely state that each node is in the Func state.

Second we introduce functions for producing lists of children. Specifically, we introduce the following two functions²:

$$\begin{aligned} \mathit{child}(N, i) &\rightarrow (i \bmod \mathit{outdeg}(N))^{\text{th}} \text{ child of } N \\ \mathit{children}(N, j) &\rightarrow \mathit{child}(N, 1), \mathit{child}(N, 2), \dots \mathit{child}(N, j) \end{aligned}$$

²Note: We are using 1-indexing for this section.

B. Formal Rendering Engine Description

And finally, we introduce a function that returns the outdegree of the provided node:

$outdeg(N) \rightarrow outdegree\ of\ N$

And finally, the shorthand notation $kids(N)$ for $children(N, outdeg(N))$.

With these additions, we can now write identity tree transducer for arbitrary arity trees as:

Ident [$?n(kids(?n))$] $\rightarrow ?n(\mathbf{Ident}[kids(?n)])$

Ident [$?n$] $\rightarrow ?n$

Let's assume the following rules are applied to a binary tree. Here is the progression of replacements that would happen on an intermediate node based on the first rule above:

1. We attempt to bind the particular LHS to the node at hand by replacing $?n$ with $this$ (where $this$ represents the current node), resulting in the LHS **Ident**($this(children(this, outde$
2. We apply the $outdeg$ method, which returns 2, resulting in LHS **Ident**($this(children(this, 2))$).
3. We apply the $children$ macro, resulting in the LHS **Ident**($this(child(this, 1), child(this, 2))$)
4. We apply the $child$ macros, resulting in the LHS **Ident**($this(a, b)$), where a and b happen to be the two children of $this$.
5. Finally, we have a LHS that can be tested for a match against the actual structure of the tree: $this(a, b)$.

Adaption for Relation-Aware Rules

Next we add a way to talk about CTS-related nodes in this syntax. If normally the syntax **state**($?n_1$) represents some node n_1 in state **state**, then we add the notation $|^{rel}n_2$ to represent the existence of some n_2 connected by a relation named rel . We use the syntax $|\emptyset$ to represent a node that does not have any relations.

8. Conclusion

So, for example, the following rule is the same identity transducer, but it is only a complete transducer for trees in which each node has an *is* relation to some other node.

$$\mathbf{Ident} [?n(kids(?n)|^{is}?other)] \rightarrow ?n(\mathbf{Ident} [kids(?n)])$$
$$\mathbf{Ident} [?n(|^{is}?other)] \rightarrow ?n$$

B.3. The Rendering Process

The rendering process breaks down into two sequential steps that this chapter will describe. The first step is *structural alignment*, which produces a new tree whose structure matches the semantic assertions made by any *are*, *are*, *if-exist*, and *if-nexist* relations in the subtree of *root*. The second step is *combination*, which replaces nodes based on *graft* and *is* relations. This second step may spawn new calls to *render* as part of that replacement process.

The pseudo-code for the implementation of *render* is thus:

```
def render(root, trees, relations):
  t <- align (root, trees, relations)
  t2 <- filter (t, trees, relations)
  t3 <- combine(t2, trees, relations)
  return t3
```

B.4. Step 1: Alignment

If-Exist and If-NExist A node related via *if-exist* to a node containing a falsey value should not emit its children, but one with a truey value should:

$$\mathbf{Align} [?n(kids(?n)|^{if-exist} ?o(FALSE))] \rightarrow ?n$$
$$\mathbf{Align} [?n(kids(?n)|^{if-exist} ?o(TRUE))] \rightarrow ?n(\mathbf{Align} [kids(?n)])$$

And for *if-nexist*, we have the opposite:

$$\mathbf{Align} [?n(kids(?n)|^{if-nexist} ?o(TRUE))] \rightarrow ?n$$
$$\mathbf{Align} [?n(kids(?n)|^{if-nexist} ?o(FALSE))] \rightarrow ?n(\mathbf{Align} [kids(?n)])$$

Are

$\text{Align}[?n(\text{kids}(?n)|^{\text{are}} ?o(\text{kids}(?o)))] \rightarrow ?n(\text{Align}[\text{children}(?n, \text{outdeg}(?o))])$

B.5. Step 2: Filtering

The second step in the rendering process filters (by marking rather than removal) relations beneath `are`-related nodes. Since `are` relations are used in to emulate the concept of foreach loops, the i^{th} and j^{th} subtrees of the `are`-related nodes have a special relationship with $i = j$ and should have no relationship with $i \neq j$. Any relations between two subtrees of differing index should thus be marked as disabled to preserve the intent of the relation for template rendering purposes.

This is necessary in the first place because the way in which relations are best crafted between `are`-related nodes is with CSS selectors that select *all* subtrees at once. For example, a selector might say "blog titles in the HTML design are the same as the blog titles in the database," or in pseudo-code selectors:

```
page | .title :is json | posts.*.title
```

This enables the author to write a single rule without having to know how many blog posts (or post design mockups) will be present at run-time. It is also necessary because it means when outdegree alignment occurs and new HTML is added to the web page, the CTS statement will apply to that new HTML as well. But it also results in $n \times m$ relations: each blog title in the HTML will be related to every single title data field in the data, as shown in Figure This is the reason for the filtering step.

We define a special filter called the **Reject-Unless** filter which rejects all relations that binding withone one subtree (exclusive of the root) unless that binding is in a particular sub-subtree (inclusive of the root). For example, the filter $RU(n_1, n_2)$ will reject any relation bound to a node in the subtree of n_1 unless that node is n_2 or in the subtree of n_2 . To say that some other n_3 's relations are filtered by $RU(n_1, n_2)$ means that relations bound to nodes in n_3 's subtree are limited to only those passing the filter. Figure 2 shows an example.

When an `are` relation connects nodes n_1 and n_2 and the aligned cardinality of n_1 and n_2 is J , the following filters are applied:

- $RU(n_1, \text{child}(n_1, j))$ for $j \in 1 \dots J$ is applied to n_2
- $RU(n_2, \text{child}(n_2, j))$ for $j \in 1 \dots J$ is applied to n_1

8. Conclusion

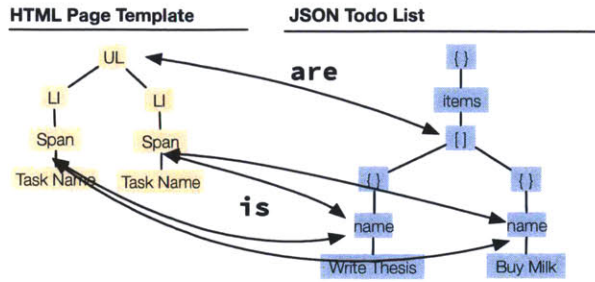


Figure 1.: When interpreting a relation-filled forest for the purposes of rendering, ambiguity occurs within the subtrees of *are* relations: which subtree relations (given multiple competing relations of the same type for a node) should be operated upon? The pruning process relieves this problem.

Reject-Unless Filters

A filter $RU(\text{reject}, \text{unless})$ accepts all relations by default. If a relation is bound to a node in the subtree of a the *reject* node (exclusive), it is rejected unless that binding also belongs to the subtree of the *unless* node (inclusive).

To say that X 's relations are filtered by $RU(A, B)$ means that relations in X 's subtree are limited to only those passing the filter.

In the figure at right, all relations pass the $RU(A, B)$ filter on X except the one connecting D to Y .

The filter $RU(A, B)$ applied to node X

Rejects relations in the subtree of X that bind to the subtree of A (exclusive) but not the subtree of B (inclusive).

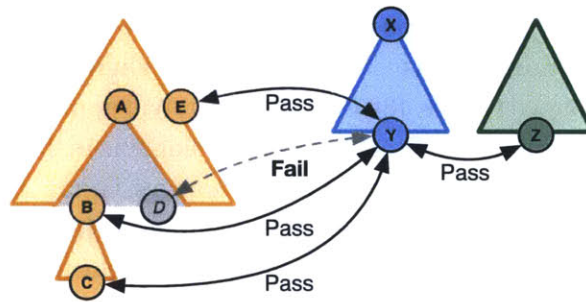


Figure 2.: The Reject-Unless filter used to prune a forest of irrelevant relations before rendering.

This permits relations between each subtree and other arbitrary trees, but limits relations between the two subtrees specifically to only those scoped to the synchronized items in the symmetric `foreach` loop.

B.6. Step 3: Combination

The final stage is combination, in which trees are combined to produce the final output tree. The *root* tree for this stage is the filtered output of Stage 2. This stage makes two complete passes over the tree. The first processes any *is* relations, and the second processes any *graft* relations.

Recall that the *is* relation is the symmetric relational version of a `print` statement. The first pass, which processes *is* relations, simply writes out the *root* tree, but any time a node has an *is* relation, it writes out the children of the related node instead of the node it transitioned to, as in the following rule:

$$\mathbf{Replace} \ [?n(kids(?n)|^{is} ?o(kids(?o)))] \rightarrow ?n(\mathbf{Ident} \ [kids(?o)])$$

Recall that the *graft* relation is the symmetric relational version of a parameterized `include` statement, e.g.: `include(subtemplate, params)`. The binding point for the inclusion is on the node participating in the *graft* relation, and the parameterization is controlled by *downtree* relations.

Thus processing the *graft* relation is similar to processing the *is* relation, except that it *downtree*-relations will be followed as well:

$$\mathbf{Replace} \ [?n(kids(?n)|^{graft} ?o(kids(?o)))] \rightarrow ?n(\mathbf{Align} \ [kids(?o)])$$

There is one important difference however because of a problem like the one solved in Step 2. Consider the following scenario: the design for a single Twitter tweet is stored in its own HTML file and every tweet on a person's Twitter page uses a *graft* relation transform the tweet from unadorned HTML to stylized HTML. This is a good example of the *graft* relation: semantically, it is similar to calling `include(tweet.html, params)` for each tweet.

But consider what happens once the transducer transitions into the children of the `tweet.html` tree for the first tweet. It begins copying all the design scaffolding and eventually reaches the first parameter—perhaps the name of the person sending the tweet. This parameterization is expressed as an *is* relation back to the caller's tree. But from the perspective of the design template's tree, there are *n* *is* relations, one per invocation.

To solve this problem, the every time a *graft* relation is followed from n_1 to n_2 , a *Reject-Unless* filter is applied to every other node that has a *graft* to n_2 . The “Unless” condition is left null, so that every relation into *other* invocations of that

8. Conclusion

template will be ignored. The scope of the filter is the processing of that particular graft subtree (that particular invocation).

B.7. Summary

The rendering engine for CTS is thus a sequence of simple tree transductions interwoven with a few offline functions to align tree structures and mark certain relations as out-of-scope given the context of uptree relations. The result of this process is a final web document stitched together from multiple source web documents. This final document is the same document that might have been produced by a web template, but the properties of the process that produced it are very different from web templating as currently practiced, enabling the new authoring methods we will discuss in the following two chapters. Before that, however, we return to the example CTS scenarios to provide the full view of these scenarios through the rendering process.

C. Rendering Examples, Under the Hood

We now revisit the example scenarios and show the intermediate transformations taking place inside the CTS engine as trees are rendered.

C.1. A Basic `is` Relation: “This this is like that thing.”

The most basic `is` example is shown again in Figure 3 along with a designation of a root node and the final output given that root node.

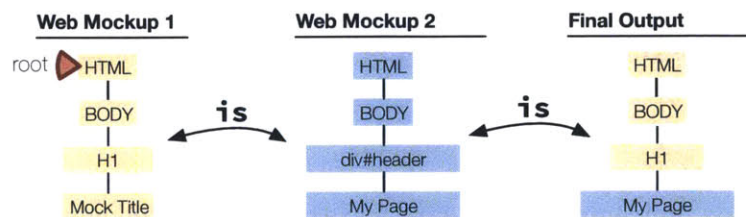


Figure 3.: The final output of a simple `is` example.

C. Rendering Examples, Under the Hood

Stages one and two simply replace exact duplicates of the root tree since this example only has *is* relations. At stage four, the following production sequence occurs.

Until the transducer hits the H1 node, it follows the identity transduction rules down the tree:

```
Replace(HTML(BODY(H1(Mock Title) | is div#header)))
```

```
HTML (Replace(BODY(H1(Mock Title) | is div#header)))
```

```
HTML(BODY( Replace(H1(Mock Title) | is div#header)))
```

Once the head is on the H1 node in the **Replace** state, the following production rule is triggered:

$$\text{Replace } [?n(\textit{kids}(?n)|^{\textit{is}} ?o(\textit{kids}(?o)))] \rightarrow ?n(\text{Ident } [\textit{kids}(?o)])$$

As a result of this rule, the following scenario will unfold, in which *?n* is written the the output tree along with the relations it participates in, but the children of *?o* are traversed into instead of the children of *?n*:

```
Replace(H1(Mock Title) | is div#header)))
```

```
H1( Replace(My Page))
```

```
H1(My Page)
```

The end result is the final tree shown in Figure 3

C.2. A graft with a nested *is*

The example of a basic graft with nested *is* is shown again in Figure 4 along with a designation of a root node and the final output given that root node.

Similar to the solitary *is* example, Stage 1 and Stage 2 of the rendering process simply copies the root tree as-is.

In Stage 3, the action begins to occur when the read head is on the following node/state:

8. Conclusion

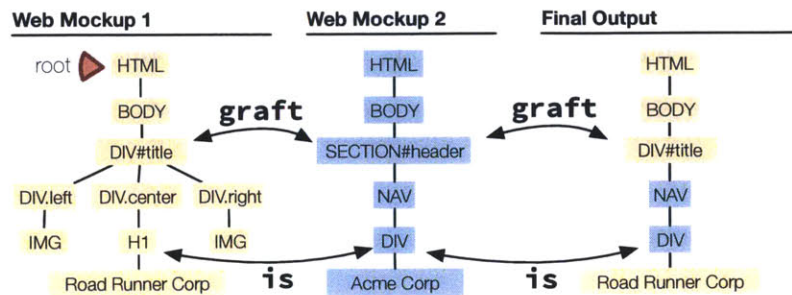


Figure 4.: The final output of an graft with nested is example.

Replace(DIV#title(DIV.left, DIV.center, DIV.right) | graft SECTION#header)

According to the graft rule

$$\text{Replace} [?n(\text{kids}(?n)|^{\text{graft}} ?o(\text{kids}(?o)))] \rightarrow ?n(\text{Align} [\text{kids}(?o)])$$

the transducer will output the DIV#title node but traverse into the children of the SECTION#header, resetting the state to **Align** for each child and prioritizing tie-breaks in case of multiple graft relations. Since there are no ties, the resulting state of production look like:

DIV#title(Align(NAV(DIV(Acme Corp) | is H1)))

The nodes rooted in the NAV are from the middle tree, labeled **Web Mockup 2** in Figure 4. Since Stages 1 and 2 are effectively no-ops, we describe the remainder of the transduction starting at Stage 3. Production processes down the tree until it hits the final is relation, upon which the children of **Web Mockup 1**'s H1 are printed instead of the DIV:

DIV#title(Replace(NAV(DIV(Acme Corp) | is H1)))

DIV#title(NAV(Replace(DIV(Acme Corp) | is H1)))

DIV#title(NAV(DIV(Replace(Road Runner Corp))))

DIV#title(NAV(DIV(Road Runner Corp)))

And the final output tree is shown at right in Figure 4.

C.3. A list of todo items

The example of a basic are scenario is shown in Figure 5 along with a designation of a root node and the final output given that root node.

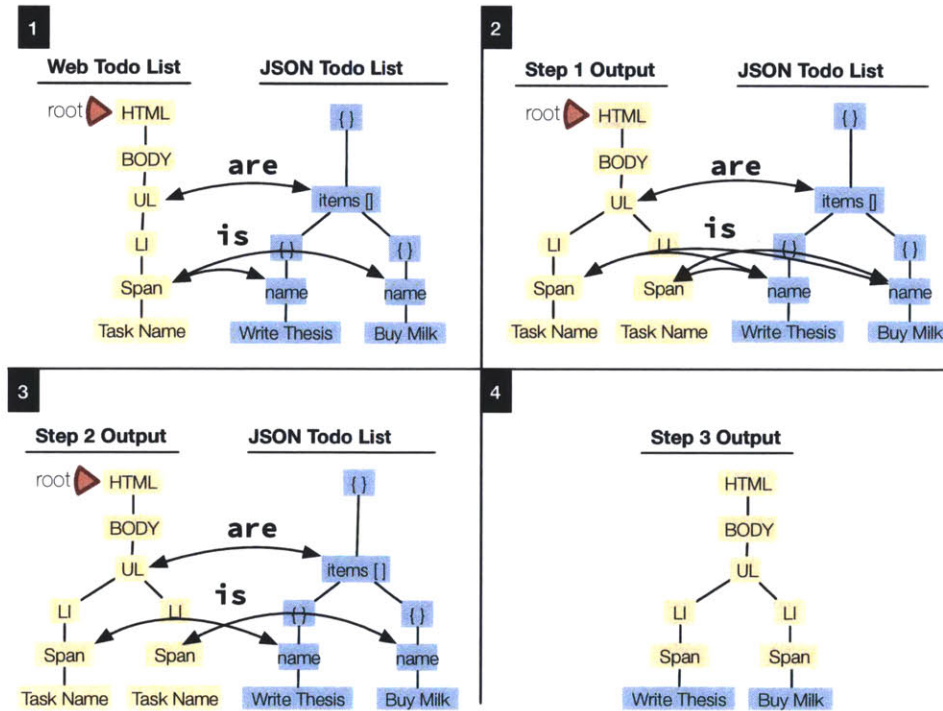


Figure 5.: Step-by-step production of a simple are example.

During Step 1, the outdegree of the UL node is aligned with the outdegree of the are-related items node. This is done as a result of the are production rule:

$$\mathbf{Align} [?n(kids(?n)^{are} ?o(kids(?o)))] \rightarrow ?n(\mathbf{Align} [children(?n, outdeg(?o))])$$

The output of this alignment is shown as Tile 2 of Figure 5.

In Step 2, a reject-unless filter is applied to each child of the are relation. This marks as disabled relations between the i^{th} and j^{th} subtrees when $i \neq j$. The output of this filtering step is shown in Tile 3 of Figure 5. Finally, the is relations result in the final tree show in Tile 4.

8. Conclusion

D. Lossless Rendering Proof

In this section, we demonstrate that rendering under CTS is lossless with respect to (1) the root tree provided to the rendering process, and (2) the relations used to make rendering decisions. These properties play an important role when we extend CTS to support dynamic read-write-compute applications in Chapter 6.

Root losslessness means that the output of the rendering process can be provided as the root to that same rendering process with the same result:

$$\text{render}(\text{root}, \text{trees}, \text{relns}) = \text{render}(\text{render}(\text{root}, \text{trees}, \text{relns}), \text{trees}, \text{relns})$$

Perhaps *root* is a data tree, or perhaps it is a fragment of design, or perhaps *root* is tree meant solely for stitching together other data and design trees. In either case, it is the tree responsible for coordinating the output viewed in the browser.

Because CTS rendering is lossless with respect to *root*, the output document is guaranteed to preserve the relational bindings necessary such that (1) it can respond to changes in other trees in the forest, (2) it can mutate other trees in the forest based on changes to itself, and (3) it can be reused by web visitors.

And because all trees that are not the primary tree are incorporated into the CTS forest by reference³, reusing the web design as a visitor (case 3 above) reduces to an extreme version of case 1. The web visitor simply replaces one or more of these linked trees with her own trees.

Losslessness through Step 1 In the first step *if-exist* and *if-nexist* relations prune regions of the *root* tree and are relations cause outdegree alignment between nodes in the root tree and node in related trees.

It is trivial to that for any *if-exist* relation against a TRUE value, any *if-nexist* relation against a FALSE value, or any *are* relation against a non-zero value, that the output tree contains both the node participating in the relation, a copy of that relation, and a copy of its children. Provided the same forest to render against, the output of Step 1 of the rendering process thus preserves all of the relevant nodes, relations, and subtrees that passed Step 1 in the original render.

³Recall that the *root* tree is the one loaded in the browser, and all other trees are included via CTS header statements such as `@html widget http://example.org/widget.html;`

D. Lossless Rendering Proof

In the particular case of HTML, we can actually go further and make the output tree preserve enough information to behave the same as *root* even when presented with *different* forests. The rendering process as described does not preserve conditional branches not taken or empty loops. But if we render all *if-exist* and *if-nexist* -related nodes anyway, hiding certain childsets rather than omitting them, then the CTS engine can leave the structure in the tree for further reuse. Similarly, an *are* relation against a node with outdegree 0 would actually be rendered as if the outdegree was 1, but this child would be hidden via CSS.

Losslessness through Step 2 Step 2 merely marks relations as being pruned rather than pruning them, so no information is removed. This pruning process can be re-run at any time and will re-mark nodes based on the structure of the forest at that moment.

Losslessness through Step 3 Like the first step, the final step of the rendering process can be shown to preserve the template structure through simple inspection. Because any node with a *graft* or *is* relation is written to the output tree along with that relation, the information necessary

Bibliography

- [1] Apache Software Foundation. The Apache Velocity Project, 2014. URL <http://velocity.apache.org/>.
- [2] Computer Science Teachers Association. K-12 computer science standards. Technical report, Association of Computing Machinery, 2011.
- [3] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. In *VLDB '12*, 2012.
- [4] Eirik Bakke, David R Karger, and Robert C Miller. A Spreadsheet-Based User Interface for Managing Plural Relationships in Structured Data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011.
- [5] David Beckett, Tim Berners-Lee, and Eric Prud'hommeaux. Turtle-terse rdf triple language. *W3C Team Submission*, 14, 2008.
- [6] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.
- [7] Edward Benson and David R Karger. End-Users Publishing Structured Information on the Web: An Observational Study of What, Why, and How. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014.
- [8] Edward Benson, Adam Marcus, Fabian Howahl, and David Karger. Talking about data: sharing richly structured information through blogs and wikis. In *Proceedings of the International Semantic Web Conference*, 2010.

Bibliography

- [9] Edward Benson, Adam Marcus, David Karger, and Samuel Madden. Sync kit: a persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the World Wide Web Conference*, 2010.
- [10] Edward Benson, Justin Meyer, and Brian Moschel. Embedded Javascript (EJS), 2014. URL <http://embeddedjs.com/>.
- [11] Edward Benson, Amy Zhang, and David R. Karger. Spreadsheet-Backed Web Applications. In *ACM Symposium on User Interface Software and Technology*, 2014.
- [12] T Berners-Lee, R T Fielding, and H Frystyk Nielsen. Hypertext Transfer Protocol - HTTP/1.0. Technical report, World Wide Web Consortium, 1994.
- [13] T Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, IETF Network Working Group, 1994.
- [14] M Bostock and J Heer. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.
- [15] M Bostock, V Ogievetsky, and J Heer. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 2011.
- [16] John Branch. Snow fall: The avalanche at tunnel creek. In *The New York Times*, 2013.
- [17] D Brickley and R V Guha. RDF Vocabulary Description Language 1.0: RDF Schema. In *W3C*, 2004.
- [18] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
- [19] Hampton Catlin, Natalie Weizenbaum, and Chris Eppstein. SASS: Syntactically Awesome Stylesheets, 2014. URL <http://sass-lang.com/>.
- [20] Tantek Celik and Brian Suda. hCard 1.0. Technical report, Microformats Wiki, 2005.

- [21] Kerry Shih-Ping Chang and Brad A Myers. Creating Interactive Web Data Applications with Spreadsheets. In *ACM Symposium on User Interface Software and Technology*, 2014.
- [22] James Clark, Steve DeRose, et al. Xml path language (xpath). *World Wide Web Consortium (W3C) Recommendation*, 16, 1999.
- [23] James Clark et al. Xsl transformations (xslt). *World Wide Web Consortium (W3C) Recommendation*, 1999.
- [24] Jeff Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, 20(9), 1987.
- [25] Dominic Cooney and Dimitri Glazkov. Introduction to Web Components. Working Group Note 20140724, World Wide Web Consortium, 2014.
- [26] Bruno Courcelle. A representation of trees by language. *Theoretical Computer Science*, 6:255–279, 1978.
- [27] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF Network Working Group, 2006.
- [28] M. Dontcheva, S.M. Drucker, M. Cohen, and D. Salesin. Relations, Cards, and Search Templates: User-guided Web Data Integration and Layout. In *ACM Symposium on User Interface Software and Technology*, 2007.
- [29] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proc. SIGMOD*, 1989.
- [30] Joost Engelfriet and Hendrik Jan Hoogeboom. Tree-Walking Pebble Automata. In *Jewels are Forever Contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83, 1999.
- [31] Jody Condit Fagan. Mashing up Multiple Web Feeds Using Yahoo! Pipes. In *Computers in Libraries*, 2007.
- [32] Danyel Fisher, Stephen M Drucker, Roland Fernandez, and Scott Ruble. Visualizations Everywhere: A Multiplatform Infrastructure for Linked Visualizations. In *IEEE Trans Vis and Comp Graphics*, 2010.

Bibliography

- [33] Michael J Fitzgerald. CopyStyler: Web design by example. Technical report, MIT Masters Thesis, 2008.
- [34] Michelle Gantt and Bonnie A. Nardi. Gardeners and gurus: Patterns of cooperation among cad users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1992.
- [35] Ferenc Gecseg and Magnus Steinby. Tree automata, 1984.
- [36] Charles F Goldfarb and Yuri Rubinsky. *The SGML Handbook*. Clarendon Press, Oxford, 1990.
- [37] Google, Inc. AngularJS, 2009. URL <https://angularjs.org/>.
- [38] Paul A Gross, Micah S Herstand, Jordana W Hodges, and Caitlin L Kelleher. A code reuse interface for non-programmer middle school students. In *the 15th international conference*, page 219, New York, New York, USA, 2010. ACM Press.
- [39] Harry Halpin, Ivan Herman, and Patrick Hayes. When owl:sameAs isn't the Same: An Analysis of Identity Links on the Semantic Web. In *RDF Next Steps Workshop*, 2010.
- [40] Pat Hanrahan. VizQL. In *the 2006 ACM SIGMOD international conference*, page 721, New York, New York, USA, 2006. ACM Press.
- [41] David Heinemeier Hansson. Ruby on Rails, 2014. URL <http://rubyonrails.org/>.
- [42] Björn Hartmann, Scott Doorley, and Scott R Klemmer. Understanding Opportunistic Design. In *IEEE Pervasive Computing*, 2008.
- [43] J Heer and M Bostock. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, 2010.
- [44] Jeffrey Heer, Fernanda Viégas, and Martin Wattenberg. Voyagers and Voyeurs: Supporting Asynchronous Collaborative Information Visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007.

- [45] Ivan Herman, Ben Adida, Manu Sporny, and Mark Birbeck. RDFa 1.1 Primer: Rich Structured Data Markup for Web Documents. In *W3C Working Group Note*, 2013.
- [46] Scarlett R Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P Bailey. Getting inspired!: understanding how and why examples are used in creative design practice. In *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 87--96, New York, New York, USA, April 2009. ACM Request Permissions.
- [47] I Hickson and D Hyatt. HTML5. *W3C Working Draft*, 2011.
- [48] David F Huynh, David R Karger, and Robert C Miller. Exhibit: lightweight structured data publishing. In *WWW*, 2007.
- [49] M Cameron Jones and Elizabeth F Churchill. Conversations in developer communities. In *the fourth international conference*, page 195, New York, New York, USA, 2009. ACM Press.
- [50] Lawrence Journal-World. Django, 2014. URL <https://www.djangoproject.com/>.
- [51] David Karger. Standards opportunities around data-bearing web pages. In *HCIR '12*, 2012.
- [52] Yehuda Katz. Handlebars JS: Minimal Templating on Steroids, 2010. URL <http://handlebarsjs.com/>.
- [53] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 1455, New York, New York, USA, 2007. ACM Press.
- [54] Stephan Kepser. A proof of the turing-completeness of xslt and xquery. *Eberhard Karls Universitat Tübingen, Tübingen, tehnline aruanne*, 2002.
- [55] Rohit Khare and Tantek Celik. Microformats: a pragmatic path to the semantic web. In *Proceedings of World Wide Web Conference*, 2006.

Bibliography

- [56] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. Supporting end-user debugging: What do users want to know? In *Proceedings of the Working Conference on Advanced Visual Interfaces*, 2006.
- [57] Scott Klemmer. For Example, March 2010.
- [58] Frederico Knabben. CK Editor, 2014. URL <http://ckeditor.com/>.
- [59] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Eurosys '13*, 2013.
- [60] Vijay B. Krishna, Curtis R. Cook, Daniel Keller, Joshua Cantrell, Chris Wallace, Margaret M. Burnett, and Gregg Rothermel. Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, 2001.
- [61] Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011.
- [62] Brian Lee, Savil Srivastava, Ranjitha Kuar, Ronen Brafman, and Scott R Klemmer. Designing with Interactive Example Galleries. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.
- [63] Håkon Wium Lie and Bert Bos. Cascading Style Sheets, level 1. Technical report, World Wide Web Consortium, 1996.
- [64] Eve Maler, Jean Paoli, C M Sperberg-McQueen, Francois Yergeau, and Tim Bray. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, World Wide Web Consortium, February 2004.
- [65] Seki Masatoshi. Embedded Ruby (ERB), 2014. URL <http://ruby-doc.org/stdlib-2.1.2/libdoc/erb/rdoc/ERB.html>.
- [66] Meteor Development Group. Meteor, 2014. URL <https://www.meteor.com/>.
- [67] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: End-user programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.

- [68] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end-user programming. In *Proc. INTERACT*, 1990.
- [69] Ben Ogle. Old school css round corners, 2009. URL <http://benogle.com/2009/04/29/css-round-corners.html>.
- [70] Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In *Implementation and Application of Automata*, pages 275--276. Springer, 2006.
- [71] Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2009.
- [72] Rasmus Lerdorf. PHP Hypertext Preprocessor, 2014. URL <http://php.net/>.
- [73] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60--67, November 2009. ISSN 0001-0782.
- [74] Jochen Rode, Yogita Bhardwaj, Manuel A Pérez-Quiñones, Mary Beth Rosson, and Jonathan Howarth. As easy as "Click": End-user web engineering. In *Web Engineering*, pages 478--488. Springer, 2005.
- [75] M B Rosson, J Ballin, and J Rode. Who, what, and how: a survey of informal and professional Web developers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.
- [76] Ryan Dahl. NodeJS, 2014. URL <http://nodejs.org/>.
- [77] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.
- [78] Roger Scowen. Extended bnf: A generic base standard. In *Software Engineering Standards Symposium*, 1993.
- [79] Alexis Sellier and Dmitry Fadeyev. LESS, 2014. URL <http://lesscss.org/>.

Bibliography

- [80] Arun Sen and Atish P. Sinha. A comparison of data warehousing methodologies. *Commun. ACM*, 48(3):79–84, March 2005. ISSN 0001-0782. doi: 10.1145/1047671.1047673. URL <http://doi.acm.org/10.1145/1047671.1047673>.
- [81] Dave Shea. CSS Zen Garden, 2014. URL <http://www.csszengarden.com/>.
- [82] Stuart M Shieber and Yves Schabes. Synchronous tree-adjointing grammars. In *Proceedings of the 13th conference on Computational linguistics-Volume 3*, pages 253–258. Association for Computational Linguistics, 1990.
- [83] Chris Stolte and Pat Hanrahan. Polaris: A System for Query, Analysis and Visualization of Multi-dimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics*, 2002.
- [84] Chris Stolte, Diane Tang, and Pat Hanrahan. Query, analysis, and visualization of hierarchically structured data using polaris. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '02*, pages 112–122, New York, NY, USA, 2002. ACM. ISBN 1-58113-567-X.
- [85] Tableau Software. Tableau, 2014. URL <http://www.tableausoftware.com/>.
- [86] Tilde Inc. Ember.js, 2014. URL <http://emberjs.com/>.
- [87] M. Toomim, S.M. Drucker, M. Dontcheva, A. Rahimi, B. Thomson, and J.A. Landay. Attaching UI Enhancements to Websites with End Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009.
- [88] Pedro Valderas, Vicente Pelechano, and Oscar Pastor. Towards an end-user Development Approach for Web Engineering Methods. In *Advanced Information Systems Engineering*, 2006.
- [89] Fernanda Vi e gas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. Many Eyes: A Site for Visualization at Internet Scale. In *Proc. Infovis*, 2007.
- [90] Amy Volda, Ellie Harmon, and Ban Al-Ani. Homebrew databases: Complexities of everyday information management in nonprofit organizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011.

Bibliography

- [91] W3C HTML Working Group. XHTML 1.0: The Extensible HyperText Markup Language. A Reformulation of HTML 4 in XML 1.0. Technical report, World Wide Web Consortium, August 2002.
- [92] Chris Wanstrath. Mustache: Logic-less Templates, 2014. URL <http://mustache.github.com/>.
- [93] Leland Wilkinson, D Wills, D Rope, A Norton, and R Dubbs. *The grammar of graphics*. Springer, 2006.
- [94] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor - Create Your Own Android Apps*. O'Reilly, 2011.
- [95] Jeffrey Wong and Jason I Hong. Making Mashups with Marmite: Towards End-user Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007.
- [96] World Wide Web Consortium. HTML & CSS Introduction, 2014. URL <http://www.w3.org/standards/webdesign/htmlcss.html>.
- [97] Stuart H. Zweben, Stephen H. Edwards, Bruce W. Weide, and Joseph E. Hollingsworth. The effects of layering and encapsulation on software development cost and quality. *IEEE Trans. Softw. Eng.*, 21(3):200--208, March 1995. ISSN 0098-5589. doi: 10.1109/32.372147. URL <http://dx.doi.org/10.1109/32.372147>.