

Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts

Sarah Chasins



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-139

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-139.html>

October 22, 2019

Copyright © 2019, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Democratizing Web Automation:
Programming for Social Scientists and Other Domain Experts

by

Sarah Elizabeth Chasins

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodik, Co-chair

Professor Björn Hartmann, Co-chair

Professor Xiaodong Song

Professor Susan Stone

Fall 2019

Abstract

Democratizing Web Automation:
Programming for Social Scientists and Other Domain Experts

by

Sarah Elizabeth Chasins

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodik, Co-chair

Professor Björn Hartmann, Co-chair

We have promised social scientists a data revolution, but it has not arrived. What stands between practitioners and the data-driven insights they want? Acquiring the data. In particular, acquiring the social media, online forum, and other web data that was supposed to help them produce big, rich, ecologically valid datasets. Web automation programming is resistant to high-level abstractions, so end-user programmers end up stymied by the need to reverse engineer website internals—DOM, JavaScript, AJAX. Programming by Demonstration (PBD) offered one promising avenue towards democratizing web automation. Unfortunately, as the web matured, the programs became too complex for PBD tools to synthesize, and web PBD progress stalled.

This dissertation describes how I reformulated traditional web PBD around the insight that demonstrations are not always the easiest way for non-programmers to communicate their intent. By shifting from a purely Programming-By-Demonstration view to a Programming-By-X view that accepts a variety of user-friendly inputs, we can dramatically broaden the class of programs that come in reach for end-user programmers. Our Helena ecosystem combines (i) usable PBD-based program drafting tools, (ii) learnable programming languages, and (iii) novel programming environment interactions. The end result: non-coders write Helena programs in 10 minutes that can handle the complexity of modern webpages, while coders attempt the same task and time out in an hour. I conclude with a discussion of the abstraction-resistant domains that will fall next and how hybrid PL-HCI breakthroughs will vastly expand access to programming.

Contents

Contents	i
List of Figures	iii
List of Tables	v
1 Introduction	1
1.1 Web Automation	1
1.2 A Note on Usable Programming	3
1.3 Contributions	4
1.4 Outline	5
2 Overview	6
2.1 Why Johnny Can't Scrape	6
2.2 Programming by Demonstration	9
2.3 DORA: Demonstrate Once, Revise Anytime	13
2.4 Helena Key Challenges	15
3 Programming Model	35
3.1 Introduction	35
3.2 Related Work	41
3.3 Formative Interviews and Design Goals	43
3.4 Helena Interaction Model	45
3.5 Algorithms	47
3.6 Conclusion and Future Work	53
3.7 Relation Selector Example	54
4 Programming Model Usability Evaluation	55
5 Skip Blocks for Failure Recovery, Incrementalization, and Handling Data Churn	61
5.1 Introduction	61
5.2 Overview	65
5.3 Semantics	71

5.4	Alternative Designs for Failure Recovery	73
5.5	Implementation	75
5.6	Benchmarks	76
5.7	Performance Evaluation	78
5.8	Related Work	83
5.9	Conclusion	85
6	Skip Blocks for Parallel and Distributed Execution	86
6.1	Introduction	86
6.2	Background	89
6.3	Signature-Based Parallelization	90
6.4	Evaluation	95
6.5	Related Work	105
6.6	Conclusion	106
7	Skip Block Usability Evaluation	107
8	Conclusion: Programming Tools for the Future of Data Science	112
	Bibliography	115

List of Figures

2.1	Programming by Demonstration for Web Automation	8
2.2	Synthesis, Programming by Example (PBE), and Programming by Demonstration (PBD)	10
2.3	The Helena PBD Tool	11
2.4	The Helena Ecosystem	11
2.5	Traditional PBD vs. Demonstrate-Once, Revise Anytime (DORA) PBD	14
2.6	Benefits of DORA and Single-Demonstration PBD	15
2.7	The Helena Design Approach	16
2.8	Synthesizer- and User-Friendly Demonstrations	19
2.9	Table Selector Synthesis Problem: Concrete Examples	21
2.10	Table Selector Synthesis Problem	21
2.11	Usability and Learnability of Helena vs. Selenium	22
2.12	Task Completion Times with Helena vs. Selenium	23
2.13	Classical Node Selector Synthesis Problem	24
2.14	Reformulated Node Selector Synthesis Problem	24
2.15	Synthesized Node Selector Accuracy Over Time	25
2.16	High-Level User Actions and Low-Level DOM Events	26
2.17	Sample Helena Program	27
2.18	Low-Level DOM Events Program	28
2.19	Helena's Bi-Level DSL	29
2.20	Interactions Replayable with Helena vs. CoScripter	30
2.21	Skip Block Parallelization Performance	33
2.22	Parallelizing Helena Programs: Programmers vs. Non-Programmers	34
3.1	Distributed, Hierarchical Web Data	36
3.2	Helena Interface and Interaction Model	39
3.3	Helena System Diagram	40
3.4	Helena Interface: Scraping Data	47
3.5	Helena Interface: Relation Highlighting	48
3.6	Relation Selection on a Sample DOM Tree	54
4.1	Task Completion Times with Helena vs. Selenium	57

5.1	A Helena Program for Scraping Authors and Their Papers	67
5.2	A Helena Program for Scraping Authors and Their Papers with Skip Blocks	67
5.3	Semantics of the Skip Block	72
5.4	Semantics of the Skip Block with Staleness Constraints	73
5.5	A Python Web Automation Script Extended with Skip Blocks	76
5.6	Data Churn and Skip Block Speedups	79
5.7	Skip Block Failure Recovery Performance	82
5.8	Skip Block Failure Recovery Performance and Data Volume	82
6.1	Hash-Based Skip Block Parallelization Performance	99
6.2	Lock-Based Skip Block Parallelization Performance	100
6.3	Skip Block Distributed Execution Performance	101
7.1	Helena's Skip Block User Interface	108
7.2	Time to Add A New Skip Block: Programmers vs. Non-Programmers	109
7.3	Skip Block Correctness for Programmers vs. Non-Programmers	111
8.1	Per-Project versus Per-Domain Collaboration Styles	113

List of Tables

1.1 Collaborations with Domain Experts: Using Web Data to Effect Social Change and Policy Action	2
4.1 Completion of Web Automation Subtasks with Selenium	58
5.1 Sample Output Dataset	68
5.2 Web Automation Benchmarks	77
5.3 Web Automation Benchmark Characteristics	77
6.1 Stable and Unstable Features of Web Automation Workloads	89
6.2 Web Automation Benchmarks	96
6.3 Web Automation Parallelization Techniques Case Study	104

Acknowledgments

Many, many wise people helped me along the path of my Ph.D., but first and foremost I must thank Josh Sunshine. Without Josh, I wouldn't have even found the trailhead. Josh, thank you for your generous mentorship from the very first moment we met, thank you for teaching me what it means to do a Ph.D., and above all thank you for setting me on this path. I am grateful to you every single day.

Thank you to Ras Bodik, my incredible advisor and the most important mentor of my life. I'm grateful beyond words to have ended up Ras's student. His mentorship and unflagging support made the Ph.D. (almost) easy; his insights made it wildly educational; the freedom he offered made the Ph.D. the funnest gig I could possibly imagine. All of which explains why my favorite advice to fellow students is "never graduate!" I know how lucky I am to have had such a positive experience, and I know it's all down to Ras's guidance—how thoughtful he is about how he forms his group, his willingness to let his students explore, his brilliance and his incredible ability to inspire brilliance in others. He is tireless and overwhelmingly creative. For the rest of my life, I'll be thanking him for the wisdom he's shared, for a thousand important and transformative and funny memories, and just for letting me do this amazing process with him. Ras, a thousand thank yous. I have loved the Ph.D. more than I can say, and I will always be grateful that I somehow got lucky enough to be your student.

Next, a tremendous thank you to Phitchaya Mangpo Phothilimthana, my academic twin. In addition to being a genius, she is a thoughtful and insightful friend, and navigating the Ph.D. at her side made every year a joy. Mangpo, it was a pleasure to stand under Sather Gate with you the day we started, and it was a bittersweet pleasure to sit next to you at our graduation. Thank you for everything. I can hardly believe we no longer share an office or lab.

I was very fortunate to kick off my Ph.D. with a collaboration with Shaon Barman. Shaon is deeply conscientious and thoughtful, and he taught me a great deal about research and life. Shaon, thank you for your patience and support through those early years, for modeling a healthy relationship with research, and for always making time for me.

I had the great good fortune to land in a healthy, supportive, and bonded research group. Shaon Barman, Thibaud Hottelier, Ali Sinan Köksal, Joel Galenson, I've thanked my lucky stars so many times that you invited me to lunch that first day. Thank you for taking me under your wings and teaching me how to do grad school. All the big and small lessons you taught me, they made the path so smooth. I am so grateful.

I owe an unfathomably deep debt of gratitude to the many social scientists and other non-technical domain experts who let this weird computer scientist hang out with them and learn about their data needs. To all of my collaborators from other fields, our collaborations have deeply inspired me, and I'm so grateful that you were willing to spend your time and energy on making these partnerships work. This thesis is a direct result of your insights and patience and hours and hours of thoughtful conversation. Thank you for all of it. It was a privilege and a pleasure to work with such a diverse, talented set of researchers. A very special and very heartfelt thank you in particular to Kyle Crowder and Chris Hess. You were so committed from the start, and you let me peek into every step of the process. I am unspeakably grateful for how you welcomed me into

your work. To Arthur Acolin, Erin Carll, Cynthia Chen, Karen Cheng, Xiangyang Guan, Hannah Curtis Hansen, Jerald Herting, Kelly Husted, Emma van Inwegen, Ian Kennedy, Nora J. Kenworthy, Adam Kirstein, Savannah H. Larimore, Claire Lawry, Jose Manuel Magallanes, Clare Ortblad, Amandalynne Paullada, Susan Stone, Rebecca J. Walter, Pengyu Yan, Logan Young, and all the others—thank you. It’s been an honor, and I’m so happy that I get to keep working with several of you.

Another tremendous thank you to my many creative, brilliant, and tireless computing and computing-adjacent collaborators. J. Christopher Anderson, Shaon Barman, Emery Berger, Michael Carbin, Maureen Daum, Joel Galenson, Liang Gong, Sumit Gulwani, Stephi Hamilton, Yen-Sheng Ho, Tim Hsiau, Tikhon Jelvis, Eunice Jun, Rene Just, Amy J. Ko, Jonathan Kotker, Jene Li, Maria Mueller, Julie Newcomb, Phitchaya Mangpo Phothilimthana, Katharina Reinecke, Jared Roesch, Paul Ruan, Cindy Rubio-Gonzalez, Caitlin Schaefer, Sanjit A. Seshia, Rohin Shah, Saurabh Srivastava, Kyle Thayer, Nishant Totla, Jeff Tsui, thank you. It was such a privilege to do research at your sides, and I hope I get many more opportunities to work with each and every one of you. Thank you for all the hours of hashing out new ideas, the insights, the thought-provoking questions, and the thought-provoking answers.

Thank you to my committee members, Björn Hartmann, Dawn Xiaodong Song, and Susan Stone. Your feedback and your probing questions were invaluable, and the day of my thesis proposal will always be one of my fondest memories of grad school. Thank you for making that process the healthy, happy, inspiring, and growth-inducing experience I loved so much.

Thank you also to the generous mentors who gave me a chance early on, Jonathan Aldrich, Christian Poellabauer, and Aaron Striegel. Especially thank you to Jonathan Aldrich for introducing me to the world of programming languages. Jonathan, thank you for developing the work that made me want to do PL in the first place, for giving me a chance after a cold email, for being such a giving and thoughtful mentor, and for showing me how broad our PL goals can be. A little later, I also had the amazing good fortune of studying under Mike Carbin, a privilege and an extremely fun time. Mike is brilliant, thoughtful, and deeply incisive, and I can’t thank him enough for a wonderful summer of his mentorship.

I could never list everyone who has given me their valuable time and advice, but a couple more mentors deserve special mention. Amy J. Ko helped me walk the first few steps along the path to HCI. I continue to envy her ability to provide *the* vital feedback after only a few minutes of context or background. Amy, thank you for your extraordinarily insightful advice and support over the past few years, for always speaking up in support of the people who need it, and for working so hard to make the community better. Kathryn S. McKinley helped me learn how to evaluate both research priorities and research community priorities, and she constantly inspires me to be a better community member. Kathryn, thank you for teaching me so much about how to mentor and support fellow researchers—and for the time and energy and creativity you’ve spent on mentoring and supporting me.

I lucked into living in two different research communities during one Ph.D., so a special thank you to the whole UW crew and especially the UW PLSE lab for adopting me so thoroughly. Maaz Ahmad, James Bornholt, Alvin Cheung, Sami Davies, Archibald Samuel Elliott, Michael D. Ernst, Yu Feng, Dan Grossman, Eunice Jun, Sam Kaufman, Martin Kellogg, Steven Lyubomirsky, Ryan

Maas, Dominik Moritz, Rashmi Mudduluru, Chandrakana Nandi, Pavel Panchekha, Joe Redmon, Talia Ringer, Jared Roesch, Amanda Swearngin, Zachary Tatlock, John Toman, Emina Torlak, Chenglong Wang, Yisu Remy Wang, James R. Wilcox, Max Willsey, Doug Woos—I was anxious about uprooting mid-Ph.D. until I met all of you and got excited instead. I'm beyond grateful for everything you taught me about computer science, metal, and veganism. It was such a privilege to pass three years in the supportive and joyful culture you've built, sharing in your warmth and generosity, getting to know these gifted and funny and deeply good people. I may not have had the official ID card, but PLSE lab feels like home. Thank you.

I already mentioned my gratitude to the older students who showed me the ropes, but they're just one slice of the brilliant, bold, beautiful crew that made up Ras's research group over the years. Thank you, Shaon Barman, Krzysztof Drewniak, Archibald Samuel Elliott, Yu Feng, Joel Galenson, Thibaud Hottelier, Sam Kaufman, Ali Sinan Koksall, Leo Meyerovich, Julie Newcomb, Phitchaya Mangpo Phothilimthana, Rohin Shah, Saurabh Srivastava, Emina Torlak, and Chenglong Wang. I loved the long conversations, the wacky ideas, the thoughtful commentary, and the thriving community that everyone built together.

I've been fortunate to have a number of conversations over the course of my Ph.D. that dramatically altered my view of what we can accomplish as researchers. I could never list all of the inspiring researchers who participated. However, I want to take a moment to note that two of the most important and transformative conversations were with Shriram Krishnamurthi and Emery Berger. Thank you for the wisdom you imparted and for taking the time to share the perspectives that I found so transformative.

And many of the beautiful, transformational conversations of my Ph.D. have been with my brilliant peers. Again, I could never list everyone, but I have to thank Mangpo, Alex, Neeraja, Eunice, Chandra, Doug, and Julie for connections, conversations, and insights that transformed my Ph.D. experience. To all of you, thank you—your generosity and openness made this process so much sweeter.

Endless thank yous to Doug Woos, who kept me sane and grounded during a year of faculty marketing and thesising. Doug, I can't believe I missed getting to know you the first time we shared a school. I'm so happy and so grateful that I got a second chance.

Thank you to Richard Wicentowski, Tia Newhall, Lisa Meeden, Andrew Danner, Charlie Garrod, Ameet Soni, and Charles Kelemen for building a department that made it so easy for an English major to fall in love with Computer Science.

I am thankful beyond measure to Dustin Trabert. Dustin, I could not have made it to this thesis without you. I don't understand how others manage to both do research and remain functional humans without a Dustin. Thank you for supporting me in everything and for putting up with me even during deadline pushes. Thank you for countless coffee dates, deadline-day treat deliveries, late-night drives, words of support, meals out of the attic and the basement, for taking care of me, for a thousand ways of making me laugh. You are magical. Thank you.

To the girls—women now, but I guess the old name stuck—thank you for always, always being there for me. Thank you for celebrating the highs and repairing the lows. I am thankful every day, knowing that I can rely on you. Thank you for the video calls and the donut celebrations and the weirdly located vacations, all the things a person needs to make a Ph.D. fun from start to finish.

Camila, Fanny, Jeanie, Michelle, you are such deeply thoughtful and deeply good people. I am so lucky to know you and unutterably lucky to call you my friends. A special thank you to Jeanie: you always have a new framing or perspective to offer, and I appreciate it every time. Thank you for talking me through every step of the way, for all the insightful conversations that help me see through new eyes.

Finally, thank you to my family. Katie, you are brilliant and warm and sweet, and you work too hard, and you're the perfect sister. Thank you for always listening and for always being willing to push back. Thank you above all to my parents, who somehow have managed to never put a foot wrong in their decades of parenting. Thank you for supporting me in every single thing I do, for never pressuring me even when I ask you to, for thirty years of the spoken and unspoken lessons that prepared me to pursue exactly what I want to pursue. I'll never be able to put in words how much I love, appreciate, respect, and admire you. I hope you know anyway. I love all of you so very, very much. Thank you.

Chapter 1

Introduction

The Helena web automation tool represents the first evidence that Programming By Demonstration (PBD) can automate large, realistic programming tasks—tasks that take programmers hours rather than minutes [15]. This dissertation describes the design and implementation of Helena and how it puts complex programming tasks in reach for non-coders and novice coders.

1.1 Web Automation

For more than two decades, social scientists from a variety of fields have argued that web data is the future of the social sciences and that they are in dire need of the large, rich, ecologically valid datasets we can extract from webpages [7, 77, 74, 76, 38, 49, 88, 21, 55, 18, 35].

Our contention is that software tools will get easier to use, so need no longer be the exclusive domain of a technological elite, and that the network society is an increasingly pervasive reality that social scientists will not be able (or want) to ignore. The information society is either around the corner, or we are already in the middle of it. Perhaps we will know which in ten years' time.

–The Internet and the Future of Social Science Research, 2008 [18]

Although these social scientists' 2008 prediction about the expansion of computer-mediated communication and its impact on human society has come true, their prediction that new computational forms of social science research will spill beyond the purview of a “technological elite” has not. Given the spate of social science vision papers in this vein, we cannot trace this failure to a lack of interest on the part of the social sciences. Rather, we must lay it at the feet of the computer science community, which has yet to provide the easier software tools the authors envisioned. With a decade-long backlog of social scientists—and other non-programmers—clamoring to use easier programming tools, we have a tremendous opportunity for the programming languages and human-computer interaction communities, a chance to build high-impact languages and tools.

Field of Study	Goal	Web Sources
Sociology	helping low-income families move to high-opportunity neighborhoods	Craigslist.com, Apartments.com
Nursing	reducing effects of perceived race on medical crowdfunding outcomes	GoFundMe.com
Political Science	increasing transparency of government agencies, governing bodies	ONPE.gob.pe, Info-Gob.jne.gob.pe, more
Civil Engineering	reducing effects of natural disasters on infrastructure and human mobility	Maps.Google.com, HoustonTX.gov, more

Table 1.1: **Web Data** → **Policy Action and Social Change**. For a sampling of my social scientist and data scientist collaborators, the team’s field, the team’s goal, and the websites from which they collect datasets with my Helena web automation tool.

Many social and data scientists face questions that they can answer by analyzing data available on the web. Over the course of the work described in this dissertation, I have worked with about a dozen teams of social scientists and other non-technical domain experts seeking web data. It quickly became clear that the demand for web data is high.

To give a sense of the data users need, consider a few of my collaborators. For my sociology collaborators, the motivating question is how to set rent thresholds for housing vouchers to help low-income families move to high-opportunity neighborhoods, and the sources of real-time data at a fine geographic granularity are rental listing websites. For my public policy collaborators, the goal is to help charitable foundations reach and communicate with new supporters, and the best data sources are social media sites. Economists look at review sites to observe how minimum wage changes affect restaurants, transportation engineers look at carpooling sites to obtain training data for driver-passenger pairing algorithms, and civil engineers look at disaster relief coordination webpages to observe how hurricanes affect cities. See Table 1.1 for a sample of our collaborators’ goals and target websites. To collect their target datasets, these are the websites they must automate.

A web automator is any program that interacts with webpages. Web scraping is the subclass that focuses on extracting data from webpages. While some applications—*e.g.*, automated webpage testing—are primarily useful for industrial purposes, scraping is useful across many fields and beyond the software industry. Its rising popularity reflects a dawning realization of the value of web data. As more people become interested in making data-driven decisions, the interest in web data grows.

While the *interest* in web data is not limited to programmers, the access to web data often is. Web automation programming is hard. To date, even medium-complexity scraping tasks require the use of traditional programming languages and libraries. The end result is a large class of social scientists and other non-coders who want access to web data but lack the programming skills to use the available tools.

The social science teams in Table 1.1 wanted a diverse array of datasets, but they had one thing in

common: all teams failed to get access to the target data via traditional programming. These teams were all comfortable writing other kinds of programs, especially data analysis scripts in Python and R. However, all teams eventually ran up against a programming or webpage reverse-engineering task that they could not complete (Chapter 3.3), which prevented them from scraping their data themselves. Existing programming tools did not meet the needs of these teams or the many other teams we interviewed. There is exciting data available on the web, and non-traditional programmers do want it, but they cannot collect it with today's tools.

Even for highly trained, experienced programmers, web automation programming is hard. In one of our user studies, only four out of 15 Computer Science Ph.D. students could complete a moderate-complexity web automation task before the one-hour cutoff time. This dissertation demonstrates that we can make web automation programming easy (Chapter 4).

The subject of this dissertation is Helena, a web automation synthesizer and its associated programming language. The goal of Helena is to put web automation programming in reach for non-traditional programmers—social scientists, data scientists, and other domain experts. With Helena, users both learn Helena and write their first successful programs in under 10 minutes. After less than 10 minutes of training, even users who identify as non-coders can parallelize a web automation program in 61 seconds.

1.2 A Note on Usable Programming

Since you are reading my dissertation, I have a brief window in which to influence you. Perhaps I will persuade you that my web automation tool is good, that you can implement it with the key insights summarized in this document, or that you can apply these insights in other domains. I hope I will. However, I would much rather persuade you to consider studying or applying Usable Programming (UP) techniques. UP is the idea of bringing together approaches from Programming Languages and Human-Computer Interaction to make programming accessible to broad, diverse audiences.

If you want to put a programming task in reach for an audience that cannot complete it with existing tools, consider taking a UP approach to designing your new language or tool. In particular, I suggest the following key focuses:

- **Machine Needs, Capabilities.** Thoroughly understand the needs and capabilities of the machine. What does a compiler or program synthesizer need in order to complete the task, and how much of the task can it do alone?
- **User Needs, Capabilities.** Thoroughly understand the needs and capabilities of the human audience. What concepts do the target users find natural or unnatural to discuss? What parts of traditional programming do they find easy and hard?
- **Fresh Problem Formulations.** Accept that your audience may need to see a different version of the problem than *you* or your programmer friends would want to see. In short, you may need to discard old problem formulations.

Chapter 2 will discuss why web automation matters and how the Helena programming language and synthesis tool meet real users' automation needs. Before our work, progress on web automation had stalled for almost a decade. Thus, Chapter 2 can also be read as a UP case study, since UP was the approach that ultimately let us make progress in this previously stagnant domain.

I believe we are still in the early stages of understanding and disseminating the principles of UP. However, I hope this dissertation gives readers a taste of the PL successes that UP can deliver.

1.3 Contributions

This dissertation describes Helena, a set of program synthesis and programming language contributions that center on the goal of empowering non-technical domain experts to write their own web automation programs. In particular, Helena contributes:

- A *DORA* programming-by-demonstration tool that accepts a single straight-line demonstration as input and produces a web automation program as output.
 - The output programs can interact with distributed data—that is, data that is split across multiple webpages. Programs can enter data, extract data, navigate between pages, interact with webpage widgets, download files, and affect server state as human browser users can.
 - The output programs can traverse and collect relational or tree-structured data.
 - Our novel *DORA (Demonstrate Once, Revise Anytime)* workflow reflects a key insight about novice programmers' preferences and skills: drafting is hard, but editing is easy.
- A *bi-level* web automation language.
 - The Helena language is a *bi-level DSL*, pairing a low-level DSL (for robust replay) with a high-level DSL (for readability and editability).
 - * The bi-level DSL is a novel approach that empowers non-coders and novice coders to program in *abstraction-resistant* domains.
 - * The Helena ecosystem includes a reverse compiler that makes the bi-level DSL approach possible.
 - The Helena language includes novel language constructs for supporting non-programmers and novice programmers in accomplishing advanced programming tasks such as parallelization and failure recovery.
- A blocks-based program editor for the Helena language.
- Formative studies of end-user needs and preferences that shaped Helena's design.
- Evaluative studies of the tool and language usability. In particular, we evaluate:

- How usable and learnable participants find our programming-by-demonstration programming model.
- How usable and learnable participants find our novel language support for tasks like incrementalization and parallelization.
- Evaluative studies of the performance of language constructs that accelerate program execution.

In short, this work contributes (i) **a tool**, the Helena web automator, which is usable and useful for a broad audience; (ii) **key insights**, summaries of the critical insights and techniques required to implement Helena; and (iii) **generalizable insights**, hints about how these same insights and techniques can be applied in other programming domains.

1.4 Outline

Chapter 2 describes the challenges of web automation programming and the approach Helena takes to shelter users from those challenges. It also highlights the key research challenges and outlines the insights that let Helena meet those challenges.

Chapter 3 describes the Helena synthesis tool in detail. It discusses the reverse compiler, relation selector, and generalizer algorithms that form the core of the PBD tool. Chapter 4 evaluates the usability of the PBD tool and its programming model.

Chapter 5 introduces a Helena language construct, the skip block, and explores how we can use skip blocks for incrementalization, failure recovery, and more. Chapter 6 details how the skip block can also be used to transparently parallelize or distribute Helena programs. Chapter 7 evaluates the usability of the skip block, especially whether it is usable by non-programmers.

Finally, Chapter 8 concludes with suggestions about next steps for end-user web automation and for usable data science tools more broadly.

Chapter 2

Overview

This chapter presents an overview of the Helena approach. We start with an exploration of why users struggle to use standard web automation programming tools. We discuss the alternative approach that we use for Helena and the design priorities that drove our techniques. We also highlight four key research challenges, and we describe how Helena meets these challenges.

2.1 Why Johnny Can't Scrape

To understand the need for more usable web automation tools, we must first understand why writing these programs by hand is so hard. Here we illustrate three of the first challenges a programmer encounters on the path to web automation.

Reverse Engineering the DOM Tree

For a glimpse at why web automation programming is so hard, consider the following webpage and a snippet of JavaScript for scraping the price of the product displayed in the webpage:



```
return document.getElementsByClassName(
    "a-color-price")[0].innerText;
```

This snippet finds the first node with the “a-color-price” class attribute and scrapes the text. Unfortunately, webpages change often—as they are redesigned, A/B tested, obfuscated, or otherwise updated. This snippet will fail if the price element ever ceases to have this class or ceases to be the first element with the class—*e.g.*, if the webpage begins obfuscating class names, if the class

name is updated, if an earlier node is assigned the same class. In order to get access to the price information over time, we need to write a much smarter selector. We need to observe the page over time until we understand the structure well enough to anticipate what features are likely or unlikely to change over time.

Reverse Engineering the JavaScript Code

Now consider another snippet with another failure case, this time for scraping the first entry in an autocomplete menu:



```
$x(searchBarXPath)[0].value = "query string";
return $x(firstEntryXPath)[0].innerText;
```

This snippet mimics how a user first enters a query into a search box, then observes an autocomplete menu. The snippet sets the search textbox’s value to the search query, at which point the textbox contains and displays the query text. Next, the snippet looks for a node at the position in the webpage’s DOM tree where autocomplete items appear and extracts its text. This snippet will never work because the autocomplete menu will never appear. The autocomplete menu is implemented with JavaScript event handlers that are called in response to DOM events. DOM events occur when the user interacts with the webpage—for instance, when mouse buttons go up or down, keyboard keys go up or down. If a user types into the textbox, DOM events fire and trigger the JavaScript that produces the autocomplete menu; if we programmatically update the textbox value, it shows the right string, but the DOM events never fire, and the relevant JavaScript never runs. To scrape the autocomplete menu, we would need to reverse-engineer the JavaScript that runs on the page.

Reverse Engineering the Server Communication

Consider one final buggy snippet, this time for adding a product to an Amazon “shopping cart.”



```
$x(secondVariantXPath)[0].click();
$x(addToCartXPath)[0].click();
```

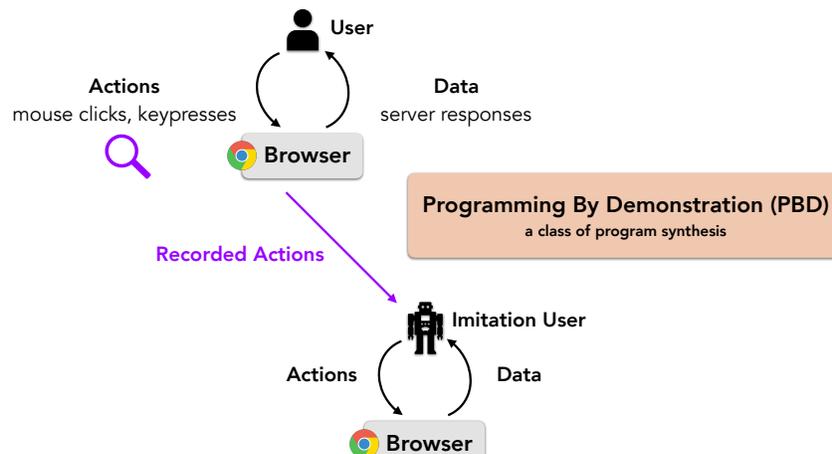


Figure 2.1: The key idea behind PBD. The synthesizer records a user’s demonstration of a task and automates the task by replaying the user’s actions.

This snippet first clicks on the button for selecting a particular product variant, then clicks on the “Add to Cart” button. Again, this snippet could fail from a failure to reverse-engineer the webpage structure or JavaScript, but this one fails for a third reason. When a user clicks on a desired product variant, the page grays out part of the page content to communicate that the webpage is not yet ready. A human viewer who is attuned to these visual cues realizes that the page is updating something and waits for the page update before clicking the “Add to Cart” button. The reason for the delay is that the webpage requests details of the second product variant from a remote server via an AJAX request, then updates the page with those details (say, a different price), when it receives a response. However, the webpage does not in fact prevent the user from clicking the “Add to Cart” button during this wait between request and response; the user can click, but it adds the original product variant rather than the last-clicked product variant. Thus, to automate this interaction, we also need to reverse-engineer the ways in which the webpage interacts with web servers, the AJAX that runs in the background on many modern interactive webpages.

DOM, JavaScript, AJAX, and the End User

To write robust web automation programs, programmers must understand several classes of browser internals—webpage DOM structures, JavaScript code that runs on webpages, and AJAX requests and responses that communicate with remote servers. Each new webpage with which a program interacts may entail fresh reverse-engineering challenges in any or all of these domains. This is challenging even for trained programmers who already know the basics of web browsers and web design, and even more so for those who lack those foundations.

2.2 Programming by Demonstration

Given that writing web automation programs requires identifying the correct DOM elements, triggering the correct JavaScript behaviors, and waiting for all requisite server responses, and given that non-coders may not know about DOM trees, JavaScript, or AJAX, how can we expect non-coders to produce these programs? The key is to observe the user.

The user may not know how to reverse engineer webpages, but if the user can perform the target interaction with the webpage themselves, a synthesis tool can observe how the user's actions affect the DOM, JavaScript, and AJAX state. The trace of observed actions and state changes can serve as a specification for the target program. From a high-level perspective, a standard human-webpage interaction is a feedback loop in which the user does some actions and the webpage returns some data. With this view of the problem, we can frame the goal of web automation as: collect the actions that a human user would take to complete an interaction, then make an artificial user that performs those same actions. This is the key idea behind *Programming by Demonstration* (PBD), a particular class of program synthesis (Figure 2.1).

Here let us briefly consider how PBD fits into the broader world of program synthesis. At the highest level, a synthesis algorithm is any algorithm that takes a specification ϕ as input and produces as output a program that satisfies ϕ —see Figure 2.2. Some readers may be familiar with *Programming by Example* (PBE), the variant in which the user provides input-output pairs and the synthesizer produces a program that maps each input to the paired output. PBD goes one step farther; the synthesizer observes input-output pairs *and* the sequence of actions that the user takes to transform the input into the output.

In the context of web automation, using a demonstration to create a web automator can dramatically simplify the user's experience. From the user's perspective, all three of the challenging reverse-engineering problems go away if they only have to provide a demonstration. In truth, we simply shift responsibility for the reverse engineering from the user to the synthesizer, and each of the three challenges requires a custom synthesis algorithm. Implementing a PBD workflow also means extracting a whole program structure from a straight-line demonstration and a set of webpages, which introduces fresh challenges. We will discuss these new challenges in [the key challenges subsection](#).

Researchers and other interested parties have been attempting to use PBD to automate web interaction tasks for decades, at least since 1997 [86, 43, 4]. First, they observed that many webpages displayed tables. In response, they developed the first table extractors [43, 4]. Later they realized that many of the important web tasks revolved around interacting with webpages, navigating between them, entering information, using widgets. They developed the first web record and replay tools for automating recorded webpage interactions [34], which facilitated tasks like webpage testing.

By 2009, PBD web automation technology still had not reached the point of producing the kinds of programs required to collect datasets like the ones described in Table 1.1, the programs non-traditional programmers need for their target tasks. Around the same time, the web started to become much more complicated. With the rise of single-page applications (SPAs) and the transition from thin-client towards thick-client approaches to web applications, pages became much more interactive. Webpages began to use much more JavaScript and AJAX, and stateful webpages became

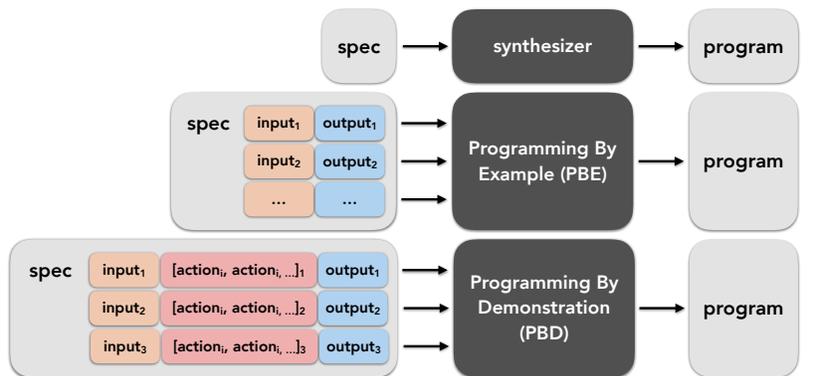


Figure 2.2: Synthesis, Programming by Example (PBE), and Programming by Demonstration (PBD). In the context of PBD, we have access not only to input and output pairs but also to the sequence of actions users take to transform the input into the output.

commonplace.

The web was no longer a map from URLs to DOM trees. It was a map from URLs to program start states. From the perspective of web PBD, the task went from reverse-engineering only webpage structure to all of webpage structure, JavaScript, and AJAX. State-of-the-art PBD had not yet reached the point of synthesizing the target program class even for static pages; even techniques for producing DOM selectors were unreliable. With pages starting to be updated more often, the DOM selector synthesis problem became even harder, and scripts failed more often. With the introduction of stateful, interactive pages, the existing tools started breaking for new reasons. No one had figured out how to make a synthesizer reverse-engineer the JavaScript and AJAX to trigger all necessary behaviors and get all necessary data from the web server. Progress on PBD web automation stalled.

The Helena Approach

Helena takes a PBD approach to web automation. Users write Helena programs by demonstrating how to complete part of the task in a standard browser—*e.g.*, collecting the first row of a dataset. The Helena synthesizer uses the input demonstration to create a program for the whole task—*e.g.*, collecting all three million rows. See Figure 2.3 for a depiction of the technologies that compose this PBD tool.

In addition to the PBD tool for synthesizing programs, the Helena ecosystem includes (i) a blocks-based program editor in which users can directly edit their programs and (ii) the Helena language, which includes specialized language constructs for advanced programming tasks like parallelization (Figure 2.4).

If you are reading this document on a computer or have convenient access to one, this is an excellent point at which to view a demo of the Helena PBD tool in action: <https://github.com/schasins/helena/wiki/Helena-Video-Demo>

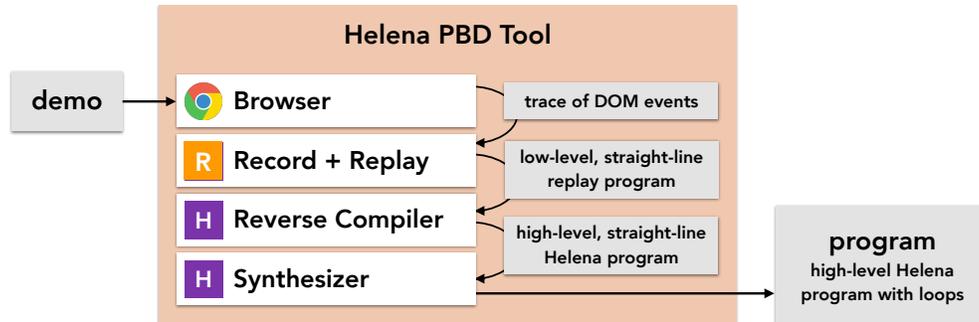


Figure 2.3: **The Helena PBD Tool.** The user demonstrates how to execute a sample of their task using a standard commercial browser. The record and replay layer collects a trace of the DOM events raised by the users actions, along with other details of the browser state throughout. From these inputs, it generates a straight-line program in a low-level language that replays the same interaction. The reverse compiler translates the straight-line program into the high-level Helena language. Finally, the Helena synthesizer analyzes the webpages with which the user interacted and adds loops to the program; with loops added, it no longer simply repeats the user’s demonstration but executes the entire task.

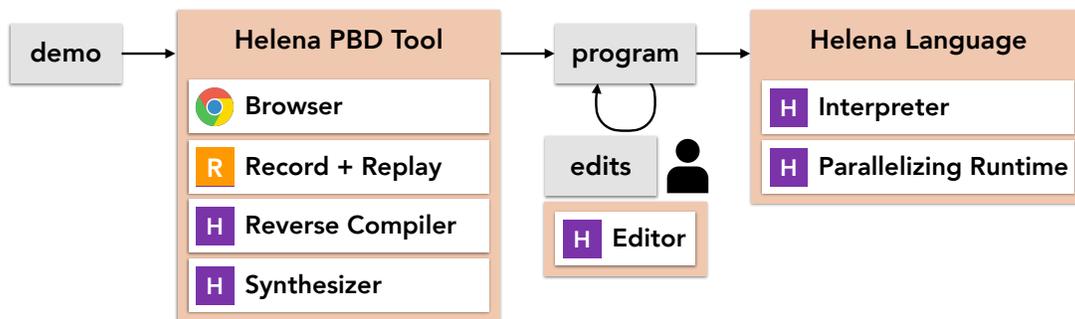


Figure 2.4: **The Helena Ecosystem.** With the PBD tool (see Figure 2.3), users draft Helena programs. Next, users see their programs in the Helena programming environment, where they can directly adapt them in a blocks-based editor. Finally, they can run the programs. The Helena interpreter includes several specialized features, including support for incrementalizing, parallelizing, and distributing web automation programs.

To use Helena, users open a standard Chrome browser. They click on the Helena icon to start the Chrome extension. They demonstrate how to collect the first row of their target dataset. If they are collecting tree-structured or hierarchical data, they collect the first row of the join of their target dataset. For instance, to collect all the papers written by the top 10,000 authors in Google Scholar, the user would load the page that lists authors, demonstrate how to collect information about the first author, click on the first author to load the page that lists the author’s papers, then collect information about the first paper by the first author.

When the user has finished demonstrating how to collect the first row, Helena analyzes the interacted webpages to identify logical relations in the page. If the user interacted with an author and Helena finds other similarly structured webpage elements that might also be authors, Helena suggests the user may want to iterate over authors. Likewise, on the webpage that lists the first author’s papers, Helena suggests that because the user interacted with one paper item, the program may iterate over all papers. It makes these suggestions by presenting a program to the user that includes these nested loops (the outermost over authors, the innermost over papers).

The tool expresses the synthesized program in the Helena language, and it displays the program in a blocks-based programming environment. At this point, users can make edits to the Helena program or, if they are satisfied with the displayed program, they can press the “Run” button to collect the full dataset. In the case of our Google Scholar example, the synthesized Helena program collects three million rows of paper data.

End-User Program Synthesis Tools

It is tempting—I hope especially after the description of Helena—to believe that end-user program synthesis tools are easy and that non-programmers should love them. Empirically, at least if we judge by adoption, this is not the case. FlashFill [24] remains as close as the program synthesis community has come to producing a mainstream tool.

Over the years, we as a community have learned a great deal about what users do not like, what they will not adopt, but we have not learned very much about what they *do* like. In part, the Helena line of work is about figuring out what non-traditional programmers do like in an end-user programming tool and what can make them successful.

The first step was to identify the patterns behind the unpopularity of prior end-user programming tools. A beautiful paper by Tessa Lau [46] lays out her view of the common pitfalls in PBD and PBE tool design. She highlights several common weaknesses, but one key point concerns how repetitive PBD interfaces can be, how long and tedious making the program specification can be. Say a user has already completed six largely repetitive demonstrations, and then the user has a tweak in mind that they feel they could express very simply in natural language—and then to make the tweak they must execute a seventh long and repetitive demonstration. Users find this frustrating, and if the program is only going to automate a small amount of work, using the tool may not be worth the effort and annoyance of providing multiple demonstrations.

Within web PBD in particular, tools that demanded multiple demonstrations ran up against user dissatisfaction. “Several users found the sequence of steps needed to construct a mashup overly complicated...it was confusing to use one technique to create the initial table, and another technique

to add information to a new column.” [53]. Even in the case of the relatively successful FlashFill tool [24], which was built around accepting multiple input-output pairs as a specification, this same complaint emerged when it was commercialized. In the course of putting FlashFill into Excel, the company studied how it works for real users and found a simple pattern. If, after providing one example, users get the right program, they are happy. If they do not, they are not happy. Across these various sources, we see one theme again and again, the fact that users dislike having to provide multiple demonstrations.

The key theme running through end users’ complaints suggests a simple question: can we invent a tool that demands only one demonstration? The natural answer if one comes from a program synthesis background is no. A single demonstration is a very ambiguous specification. Say we have demonstrated how to collect one of the papers we want a program to collect, the first paper by a given author. Do we want a program that collects all papers by all authors? One that collects the first paper by each author? All papers with more than x citations? All papers that include a given word in the title? Any of these programs would be consistent with the single-demonstration specification. How can the synthesizer distinguish between the program we want and all the others?

Key Observation

Drafting is hard. Editing is easy.

Helena’s design reflects a realization about what makes traditional programming difficult for non-programmers and novice programmers. Our social science collaborators consistently reported that drafting programs is hard for them. Many of them do in fact have some programming background—typically up to one semester of training in Python or R for data analysis. However, even in the context of data analysis scripts, they reported that they rarely sat down and wrote a new one from scratch. Many described experiencing aspects of *blank page syndrome*, finding it intimidating to face an empty text editor. Rather than write from scratch, they start with a script from a past data analysis project or from a course, then tweak it to meet their current needs. This is the approach that works for them. The key insight: drafting is hard, but editing is easy.

2.3 DORA: Demonstrate Once, Revise Anytime

Helena’s design is built around moving away from the traditional PBD approach of demanding large numbers of disambiguating demonstrations. In traditional PBD, the user provides many demonstrations as input to the synthesizer, which then produces a program. The user debugs by running the program and observing the output. If the program output is incorrect, the user goes back and provides another demonstration. If the program output is correct, the process is done, and the user exits the feedback loop. See Figure 2.5 for a standard PBD workflow.

Given the usual assumptions about end-user programmers (especially that they cannot read programs) and the standard design of synthesis languages (designed for conciseness rather than human usability, often unreadable even for trained programmers), traditional PBD design makes sense. It lets us build tools around the tasks the audience can do, demonstrating processes and

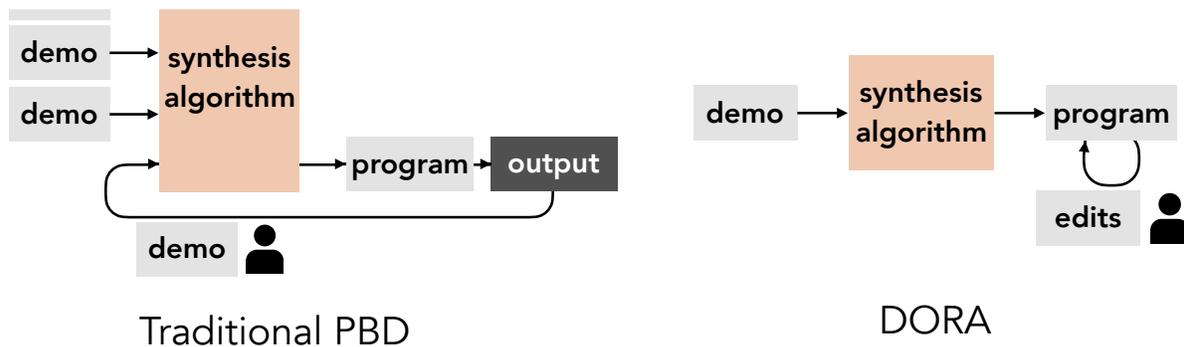


Figure 2.5: Traditional PBD vs. Demonstrate-Once, Revise Anytime PBD. In traditional PBD, the user provides multiple demonstrations and debugs the synthesized program by observing the output and providing additional demonstrations if the output is incorrect. Typically the program is expressed in a language optimized for synthesis rather than for human readability. In DORA PBD, the user provides one demonstration and debugs by directly editing the program in a readable, learnable language.

inspecting program outputs. However, this design also results in the multi-demonstration interfaces that users find so frustrating.

In place of the standard PBD workflow, Helena uses a novel **Demonstrate Once, Revise Anytime**, or DORA, workflow. The user provides a single demonstration. The synthesis algorithm uses the demonstration as input to produce a draft program, a program that is close enough to the user’s intent and readable enough that the user can then directly edit it to meet their needs. This suggests a few key design goals for the language and synthesizer. First, the output program must be expressed in a learnable language, a language that the user can read, understand, and edit. Second, since some edits are harder than others—for instance, adding a loop is harder than removing a loop—the synthesizer should aim not to produce the *likeliest* program but rather the *most editable* program. Figure 2.5 illustrates the DORA workflow. Look back at Figure 2.4 to observe how Helena’s workflow uses DORA.

DORA Advantages

We have touched on the key motivation for transitioning to a DORA style, namely that a single-demonstration interface makes users happy and successful. However, the single-demonstration approach comes with several other benefits as well, as illustrated in Figure 2.6.

First, DORA reduces the constraints on the builder of the synthesis algorithm and puts much larger programs in reach. For a traditional synthesizer to be useful, it must be able to synthesize all possible useful programs, to reach all points in the program space. In contrast, a DORA synthesizer can be useful even if it only reaches a subset of all possible programs, points in the space from which the user can edit the programs to reach other points. This has the potential to dramatically reduce the space of programs a synthesizer must search. Enormous search spaces that grow exponentially with the size of the program are the critical factor that prevent traditional program synthesis from

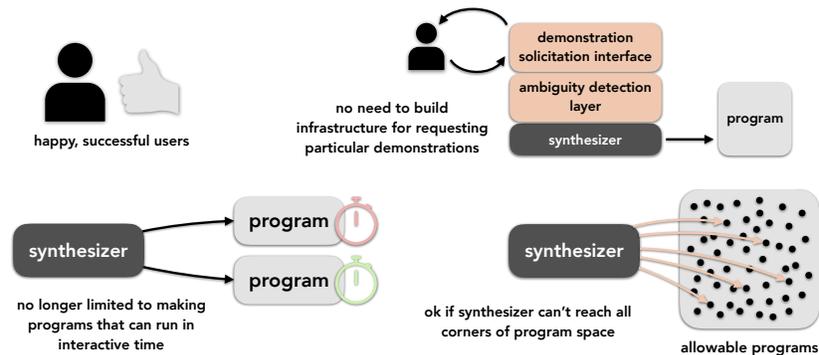


Figure 2.6: Benefits of a single-demonstration or DORA synthesizer design. Our reading of the literature suggests users object to needing to provide more than a single demonstration, whereas they are happy and successful with tools that require only a single demonstration or example. Requiring only a single demonstration also reduces the need for demonstration solicitation infrastructure. By pairing a drafting tool with a learnable language, DORA also expands the classes of programs for which we can apply PBD. For instance, long-running programs come into range, if users can debug them by reading them in a learnable language, rather than needing to run them and inspect the output. Finally, allowing users to directly edit programs (because they are expressed in a learnable language) dramatically reduces the constraints on the synthesizer. In contrast to a traditional synthesizer, which must be able to generate any program in the space of programs, a DORA synthesizer only needs to be able to generate a subset of programs—for each input, a program close enough to the target program that the user can tweak it to match their needs.

scaling to large programs, so this change holds the key to making program synthesis practical for large, real programming problems. With this approach, Helena represents the first evidence that PBD can be used to automate large programming tasks—tasks that take experienced programmers hours rather than minutes.

Second, DORA reduces the supporting infrastructure that a tool builder needs to provide. For instance, we no longer need to build up a layer for detecting ambiguity or an agent that solicits disambiguating demonstrations from the user.

Finally, DORA expands the class of programs that can be produced via PBD. Traditional PBD demands that a user must be able to debug the target program by inspecting its output. For practical purposes, this means the program must run very quickly, and the output must be small. With a DORA workflow, the user first debugs by inspecting the program itself. Thus programs, like scraping programs, that can run for hours or days come into range. Likewise, programs with large outputs, too big to inspect by hand, come into range.

2.4 Helena Key Challenges

To make a DORA tool, we must build: (i) a usable program drafting tool, (ii) a learnable language. Naturally these components may be useful and desirable even outside the context of a DORA

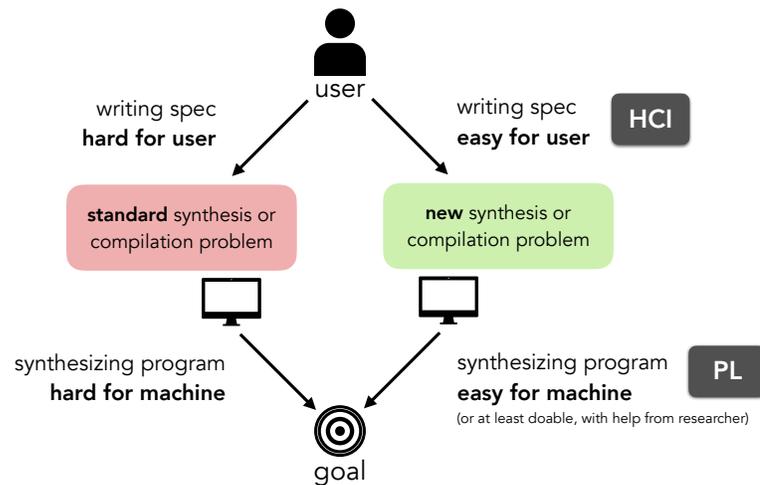


Figure 2.7: Our approach. Many of the challenges discussed in this thesis shared a common form. We have a goal and a standard problem formulation. If the human user can produce the inputs required by the problem formulation (*e.g.*, half a dozen demonstrations) and the computer can produce the target output (*e.g.*, a scraping program) from the user-provided inputs, we reach the goal. We want to reach the point where both the human and the computer find their portions of the task easy. For the problems discussed in this dissertation, the standard problem formulations made the tasks too hard for the human, the computer, or both. For the human, this usually meant a programming tool was too hard to use. For the computer, this usually meant a synthesis algorithm could not scale to large programs or produced brittle programs. Our approach was to develop new problem formulations that achieve the same goals but ask for inputs that (i) human users find easy to give and (ii) a synthesizer or compiler can convert into the target output.

workflow, but we focus here on how they can work together. The remainder of this overview chapter describes how we designed the Helena drafting tool and the learnable Helena language. In this overview chapter, we will focus in particular on four key, illustrative research challenges:

- Synthesizing complex programs from a single demonstration.
- Making programs robust to webpage changes.
- Making programs both robust and human-readable.
- Facilitating end-user parallelization.

For each goal, we take a shared approach. We come to a standard program synthesis or compilation problem. Typically the initial problem formulation, since it constitutes a research challenge, is either hard for the user or hard for the machine, maybe both. From the user’s perspective, it may be hard to write the specification for the synthesizer or the program for the compiler. From the machine’s perspective, it may be hard or time-consuming to turn the user’s input

into the correct program. We throw away the old problem formulation and develop a new framing of the problem based around making it easy for the user to write the specification and also easy for the machine to turn that input into the program the user needs—or at least doable, with thoughtful synthesis algorithm design. See Figure 2.7. Typically, the key breakthrough is developing the novel problem reformulation.

At the highest level, understanding the needs and capabilities of the user is about taking an HCI approach and understanding the needs and capabilities of the synthesizer is about taking a PL approach. Because developing problem reformulations that meet both user and machine needs is the key step, we need both HCI and PL expertise at the same table, working together. This brings us to the idea of usable programming and bringing programming to a broader audience.

Here I lay out the strategy I advocate for meeting a new programming need:

- **Identify User Needs and Capabilities** Study what users want their programs to do. Study what is hard or easy for the user to express. What kinds of ideas can they communicate naturally and fluently when they discuss the tasks on which they are working? What is hard and easy for them about traditional programming?
- **Identify Programming Tool Needs and Capabilities** Study what inputs programming tools need in order to produce classes of output programs. Come in to the design process with knowledge of what is hard and easy for synthesizers, compilers, and any other programming tool the target users might need. Know what these tools can do for the intended audience.
- **Focus on the Problem, Not the Solution** This final and most controversial piece of advice is about de-emphasizing fancy new solutions. Instead of attempting a new solution to a standard problem formulation, identify the end goal, then reformulate the problem until it meets the same goal but is amenable to simple, elegant solutions.

Here let us take a moment to consider the final prong of the strategy and why I include it here. In order to build a programming tool, typically, one must be a programmer. Builders of end-user programming tools thus find themselves in an unusual position. They are building tools for users with much less programming background, perhaps even less shared vocabulary. This is quite a different experience than they might have building standard programming tools. The tool builder and all of their programmer friends have substantial shared training and background, and they may all be inclined to tackle a given problem in a particular way. This can make it tempting to push on *that* solution, on making that solution a little better. However, this may not be the right approach *for the tool's target audience*. In cases where we want to make tools that work for non-coders, people who do not share the same training and background, it may be better if we throw away that old solution and make a new problem formulation that is more suited to the kinds of information they are comfortable contributing.

The strategy described above shaped our solution to each key Helena research challenge.

Challenge 1: Synthesizing complex programs from a single demonstration

The synthesizer design process began with identifying the kinds of data that social scientists and data scientists need. A formative study revealed users needed: (i) **Multi-Load Data**: They were not going to one webpage and extracting an entire dataset from that single page. To access the data they wanted, they were doing all the same things one does during daily web browsing activities—clicking on links and other webpage elements, interacting with widgets, entering data into forms. (ii) **Hierarchical Data**: They wanted tree-structured data—*e.g.*, not just a table of authors, but a table of authors and a table of papers with the links between authors and papers maintained.

The need for hierarchical data is especially challenging, because collecting hierarchical data means the synthesizer must be able to produce programs with nested loops. (In our authors-papers example, the outer loop iterates over authors, and the inner loop iterates over all papers by each author.) Because of how dramatically loops increase the space of programs to search, most state-of-the-art synthesis tools do not produce programs with loops. The space of programs with no loops is already frequently big enough to prevent scaling to large programs. The space of programs with singly nested loops is even larger, enough to discourage most tools from searching this space, even if their algorithms can theoretically produce programs with loops. If we expand the space again to programs with nested loops, the space is massive.

Making a synthesis algorithm that can successfully search the huge space of programs with nested loops is a long-standing research challenge, so here we reach a fork in the road. Traditionally at this point, we hunker down to build a better, faster synthesis algorithm. However, the research community has been pushing on this direction for a couple decades without reaching the class of program structures these users needed. At this point we asked if there is another path, another way to produce the programs users need.

For any given task a user wants to automate, there are many different demonstrations they could offer. For our authors-paper task, the user might collect a few of one author’s papers, press the browser back button to return to the list of authors, collect information about an author. Or the user might collect information about an author, click on the author link, then collect information about the author’s first paper. If we allow both of these demonstrations, there is little a program synthesis algorithm can do except search the space of Helena programs until it finds one that produces the sample data that the user provided. On the other hand, if we ask users to only give demonstrations like the second one, we can make the synthesis process much simpler. Notice that for the second demonstration, the key transformation required to turn the user’s straight-line sequence of actions into the final program is identifying where to insert loops (and over what data). In short, one of these demonstrations makes the synthesis problem hard, and one of them could make the synthesis problem easy. The key here is that if we can limit the user a little, we can make the synthesis design problem much easier.

Within the space of demonstrations a user might offer, there is a class of synthesizer-friendly demonstrations. For various subclasses of the synthesizer-friendly class, we know how to build synthesizers that can efficiently produce output programs from the input demonstrations. For any space of programs outside of the class or even partially outside the class, we simply do not know how to build that efficient synthesizer.

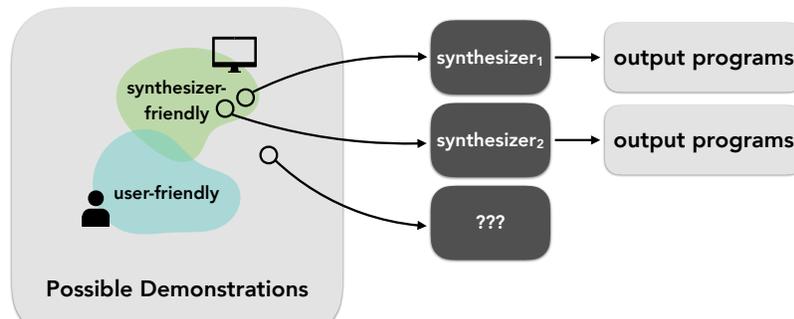


Figure 2.8: **Synthesizer- and User-Friendly Demonstrations.** For *synthesizer-friendly* inputs, we can write specialized synthesizers. For those outside of the synthesizer-friendly class, search-based techniques are our only feasible strategy. *User-friendly* inputs are simple enough that a user can understand what they need to do and does not find the tool frustrating. To produce a usable tool that scales to large programs, we must select a class of inputs at the intersection of the synthesizer- and user-friendly classes.

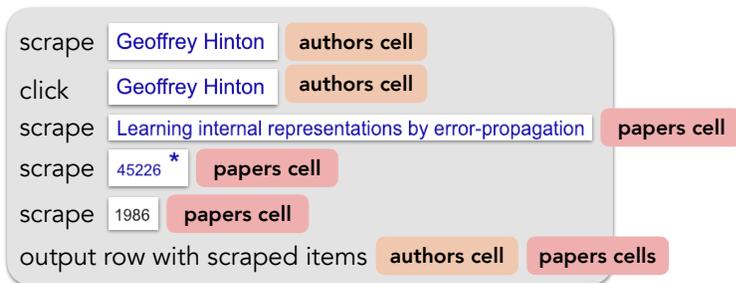
Naturally, if we allow synthesizer friendliness to be our only design constraint, we could easily end up with a baroque set of requirements for our users to learn. We have already seen that the multi-demonstration interfaces that made disambiguation easy—a synthesis-friendly approach—ultimately made users frustrated. If the restrictions we place on input demonstrations become too complicated or difficult, we can no longer expect the human user to produce inputs from that class. We must find a set of inputs at the intersection of the synthesizer-friendly and user-friendly classes (Figure 2.8), something we expect the user can comfortably provide.

To elicit synthesizer- and user-friendly inputs, Helena makes a contract with the user. The user must:

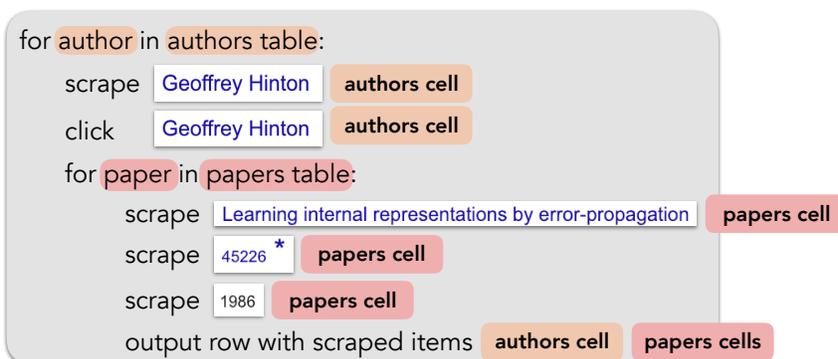
- Execute the first iteration of each loop
- Ordered from outermost to innermost loop

Naturally, Helena’s tutorials and documentation do not use this same terminology to describe the contract, but these are the key constraints on the input demonstrations. The user must execute the first iteration of each loop—collecting the first row of each table of interest. The demonstration must start with the outermost and descend towards the innermost loops. We are helped by the fact that the structure of web navigation enforces this second requirement.

The contract with the user limits the inputs the user can give the synthesizer, gives us much more information about each input demonstration, and ultimately lets us dramatically simplify the synthesis problem. We are left with the new but simpler problem of where to insert loops.



Consider the straight-line program above. Where do loops belong in this program? Let us start by assuming we know about all tables of data that interest the user—in this case, the authors table and papers table. We can label each action the user took according to whether it uses any cells from either table. (See the orange and red boxes next to the statements above.) From here, we can insert a loop over each table, starting the loop before the first use of any of the table’s cells and ending it after the last use:



The assumption that we know all tables of interest is a big assumption, given that our input demonstration has given us only a sequence of actions on an assortment of nodes from one or more webpages. On any given webpage, which interacted nodes are part of a table and which are not?

Consider the webpages in Figure 2.9, with interacted nodes highlighted in red. The user may have scraped, clicked on, or typed into the interacted nodes. To a human observer, it will probably be clear which nodes are part of a logical table that has multiple rows on the same webpage. In the Google Scholar page, all highlighted nodes appear to be part of the first row of a papers table. In the Craigslist page, none of the nodes appear to be part of the first row of a listings table, since no other listings are visible on the page. In the Twitter page, a subset of the nodes appear to be part of the first row of a tweets table. At a high level, Helena uses the same intuitions as a human viewer.

The *table selector synthesis problem* takes as input a record-time webpage W and a set of interacted nodes $I\{n \in W \mid \text{interacted}(n)\}$. With these inputs, the synthesizer must produce a table selector $s.\mid s(W)_{0,*} \cap I \mid \text{maximized} \wedge \text{rows}(s(W)) > 1$, as illustrated in Figure 2.8. We want the selector to produce a table with more than one row because the trivial table that has a single row of all interacted nodes is not the best answer. (Consider how the trivial single-row table would work in the Twitter sample webpage). The output table should have multiple rows but also include as many of the interacted nodes as possible in the first row.

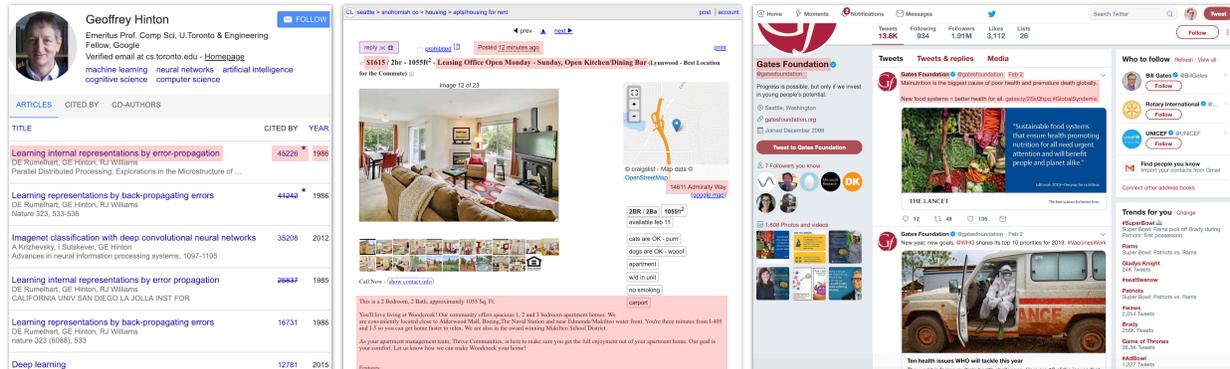


Figure 2.9: Red-highlighted nodes represent nodes with which a user interacts during a demonstration. Any or all of them may be elements of the first row of a table. This information is the input to the table selector synthesis problem.

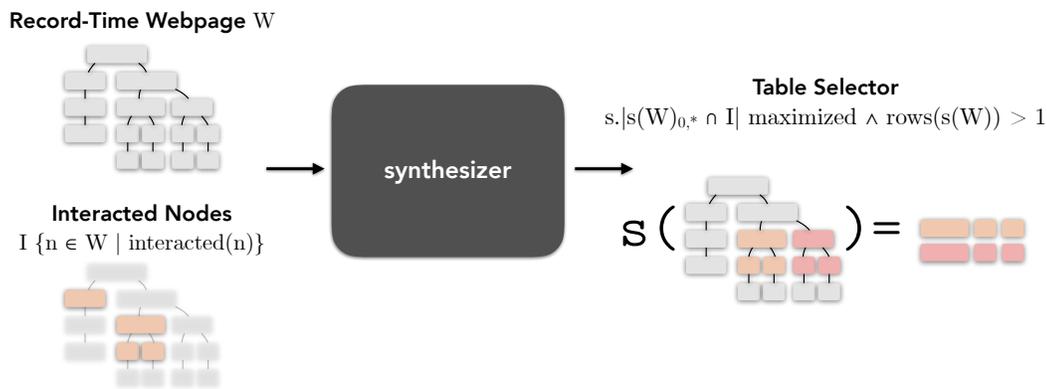


Figure 2.10: **Table Selector Synthesis Problem.** Given a record-time webpage and a set of interacted nodes, the goal of table selector synthesis is to identify which nodes are part of the first row of a table and to generate a selector that extracts the table.

The first task is to find the appropriate cells to include in the table—we identify the appropriate table output before we synthesize the table selector, a separate problem. This is not traditional relation extraction, in which the user has labeled multiple rows of data in an unstructured document; rather, this is the process of identifying iterable objects that should be treated together. Identification of iterable objects should typically leverage domain-specific patterns. Here, Helena uses the fact that web best practices demand the use of templates that produce repetitive subtrees in the webpage. Logically similar information is thus presented in structurally similar ways.

Helena begins by guessing that all interacted nodes in a given webpage are part of the first row of a table. If the nodes produce a subtree structure that can be found elsewhere in the webpage, we have found multiple rows of the table, and we can pass these on to the next stage, relation extraction. On the other hand, if no other parts of the webpage share a template, Helena backs off to smaller

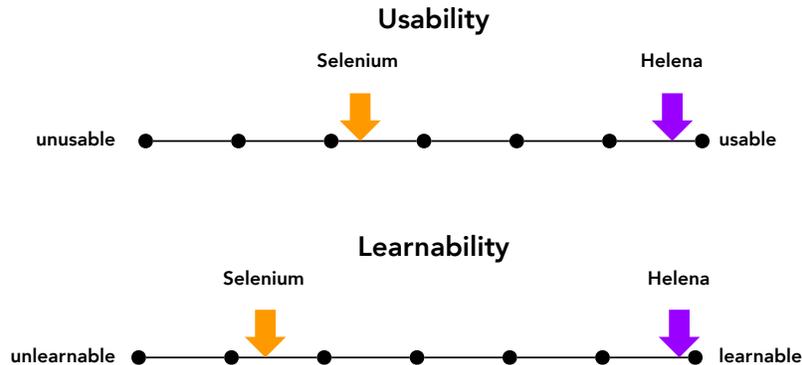


Figure 2.11: **Usability and Learnability.** Usability and learnability ratings for Helena and Selenium on 7-point Likert Scales.

and smaller subsets—beginning with all subsets of size $|I| - 1$, then all subsets of size $|I| - 2$, and so on. If we eventually find a subset that shares a template with another page component, this becomes the input to the relation extractor. If we never do, we conclude there are no relevant logical tables on the page, and no loop is inserted into the output program for the current webpage.

From the machine perspective, introducing this formulation of the problem shifts Helena from having to search the space of all programs in the target language to the still difficult but much more constrained table selector synthesis problem. The Helena interface is also a dramatic shift from the user perspective. The user in the traditional programming realm has to reverse-engineer webpage internals for one or more webpages for each new scraping program. With Helena, the user has to learn and master a demonstration contract once.

This is how we achieve a highly usable tool. Comparing traditional web automation with the Helena approach, we find Helena *is* dramatically more usable and even more learnable (Figure 2.11). It also makes users more effective. See Figure 2.12 for the change in completion rates and times. Even among computer scientists, we leap from 27% completion with traditional programming to 100% completion with Helena. All users went from never having heard of Helena to having learned the contract and produced the target output program in under 10 minutes.

Challenge 2: Making programs robust to webpage changes

To understand this challenge, we need some background on webpages. One constructs a webpage by building a tree of nodes. Nodes have text or image contents and a variety of other attributes—font, background color, x-coordinate—mapped to values.

Unfortunately for our purposes, webpages change all the time. They may change for many different reasons: webpage redesigns, A/B testing, obfuscation, even straightforward cases of fast-changing content like breaking news or social networks’ user-contributed content. Further, they may undergo multiple classes of changes: node content can change, the structure of the tree can change, the values of node attributes can change, or the set of attributes associated with a node can

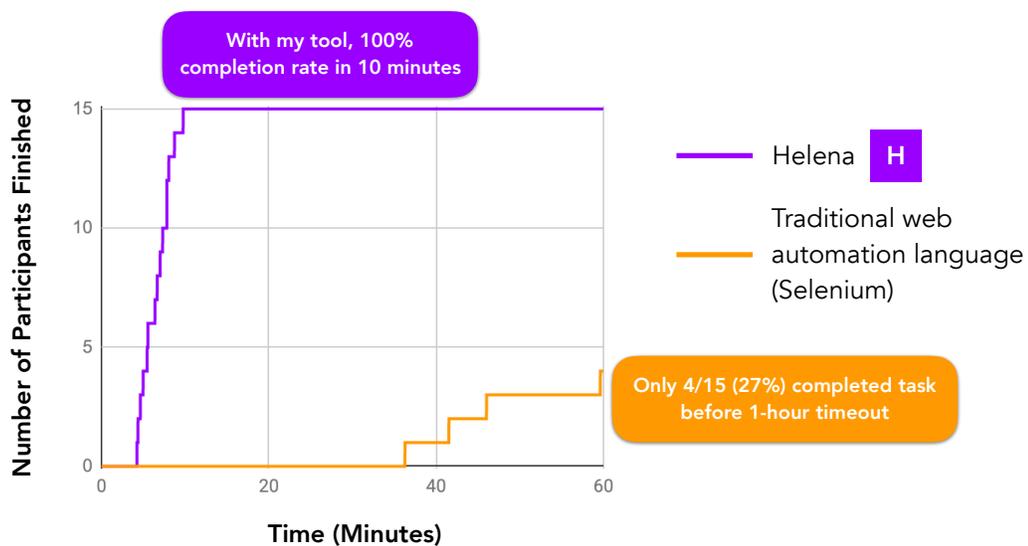


Figure 2.12: **Task Completion Times.** Time to complete a web automation task with traditional programming versus with the Helena ecosystem of tools. With Python and Python’s Selenium library, only 27% of users complete the task. With Helena, 100% of users go from never having heard of Helena to having successfully written the target program in under 10 minutes.

change. A web automation program must be able to find its target nodes even in the face of these changes.

Here we must draw a distinction between deterministic replay, with which some readers may be familiar, and generalizable replay. In the context of web record and replay, we should expect that by the time we do a replay of a recorded script, the input webpages will have changed. If we aim to do a deterministic replay, the correct move at this point is to throw away the new webpages, overwrite them with the record-time webpages, then execute the recorded program and ultimately return the recorded answer. This makes sense if we are, say, attempting to reproduce a buggy trace. In contrast, Helena’s record and replay aims to do generalizable replay. We need to be able to run Helena programs on the new inputs, and we need to be able to return the new answers. This raises the question of how Helena programs can find their target nodes in updated replay-time input webpages.

The traditional web record-and-replay approach is to assume that a few key attributes—perhaps the text, perhaps the ID—will remain stable. The problem is that they are not in fact stable.

The classical node selector synthesis problem, as pictured in Figure 2.13, takes the rerecord-time webpage W and the target node $t \in W$ as input, then synthesizes a selector $s.s(W) = t$, a selector that identifies the correct node in the webpage we have already observed. We initially attempted to solve this problem in the traditional way—especially focusing on redesigning the Domain-Specific Language (DSL) in an attempt to capture the features that would make the output selectors robust. However expressive the DSL, this problem formulation did not result in robust selectors.

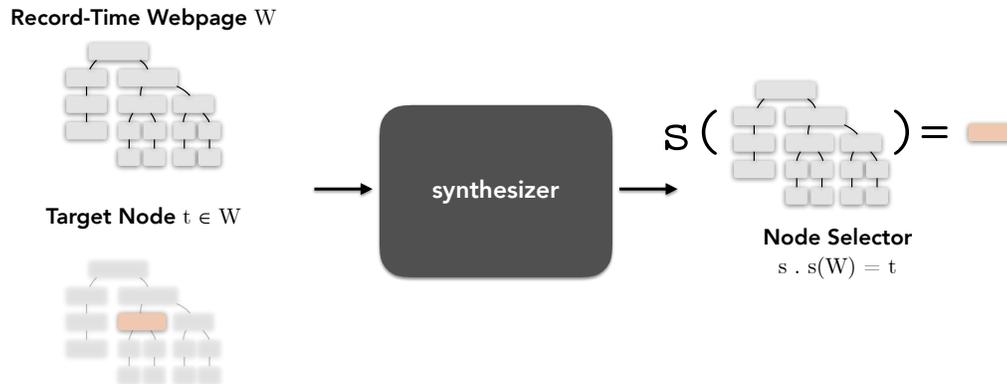


Figure 2.13: **Classical Node Selector Synthesis Problem.** A traditional formulation of the node selector synthesis problem. Given a webpage and a node in the webpage, synthesize a selector that finds the node in the webpage.

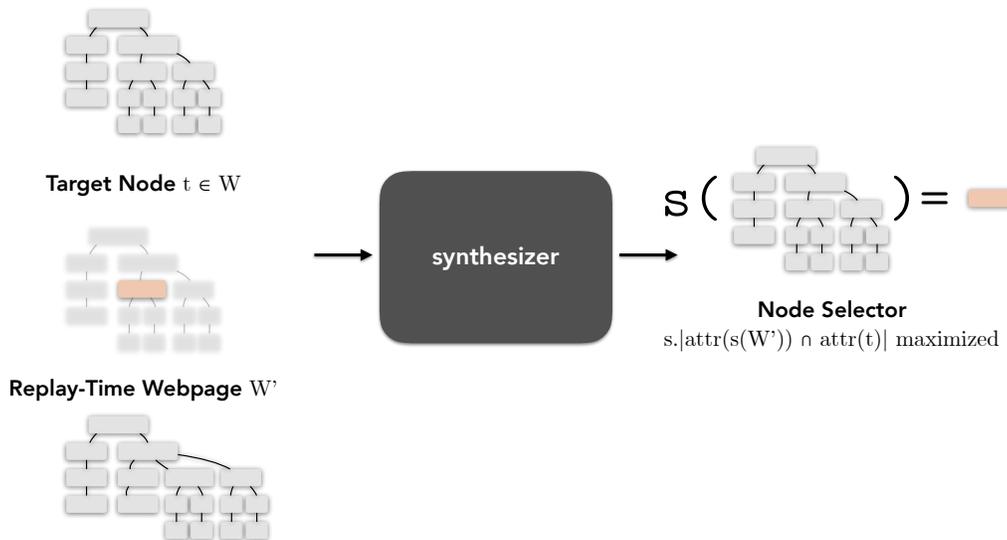


Figure 2.14: **Reformulated Node Selector Synthesis Problem.** Our novel formulation of the node selector synthesis problem. Given a webpage, a node n in the webpage, and a variant W' of the webpage, synthesize a selector that finds the node in the webpage W' that is most like n . By deferring synthesis until replay-time, we can take advantage of knowledge about how the webpage has actually changed since record-time, rather than attempting to anticipate changes at record-time.

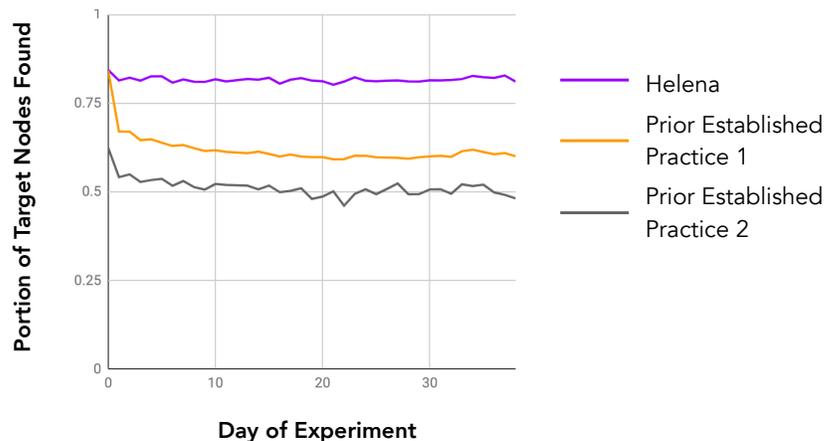


Figure 2.15: Synthesized node selector accuracy over time. For selectors synthesized on day 0, performance over the course of a month.

The solution was to discard the classical node selector synthesis problem and replace it with a new problem formulation, a deferred synthesis approach. The deferred synthesis approach takes advantage of the fact that at replay time, we actually know how the webpage has changed, and we can leverage that additional information. The new problem formulation accepts the record-time webpage W , the target node $t \in W$, and the replay-time webpage W' as inputs. The synthesizer then produces a selector s that maximizes $|\text{attr}(s(W')) \cap \text{attr}(t)|$. Figure 2.14 illustrates this deferred synthesis problem formulation.

At record time, Helena records all available attributes of the target node—the ones that are available by default via browser APIs, but also many computed attributes and custom-defined attributes. The end result is a map from 300 or more attributes to their associated values. At replay time, Helena scores all webpage nodes according to how many attributes they have in common with the record-time target node. Finally, Helena selects the node that has the highest score.

The end result is that Helena programs are much more robust to webpage changes than the prior state of the art. Figure 2.15 shows how Helena selectors compare to other established practices over the course of a month. The move from synthesizing node selectors at record-time to synthesizing them at replay-time produces better results even for the first few days, when we should expect few webpage changes.

For the rest of this dissertation, we will mostly set aside challenges like this one that we solved at the level of the record-and-replay tool. One way to decompose the PBD task is to break it into: (i) a record phase, in which the tool records the user’s actions during a demonstration and synthesizes a program that repeats those same actions and (ii) a generalization phase, in which the tool takes the straight-line replay program as input and incorporates other information (e.g., analysis of webpage structures) to synthesize a program that completes the whole task. Helena uses our custom Ringer replayer for the record phase [8]. The fact that we follow the record phase with a generalization phase imposes many interesting constraints on the design of the record tool; this is one of the factors that drove our decision to implement our own novel replayer. However, some of

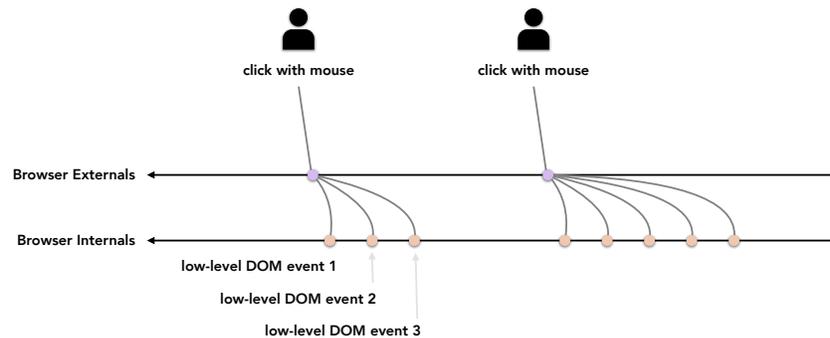


Figure 2.16: When a user performs an action on a webpage, it triggers the webpage to raise DOM events. The browser responds to DOM events with default webpage or browser state updates, and the webpage designer can use event handlers to perform webpage-specific state updates. The sequence of DOM events raised in response to an action may vary, even if the human user perceives it as the same type of action.

the key challenges—e.g., finding webpage nodes even as the webpages change—are universal web record-and-replay challenges and are not dramatically altered by the needs of the generalization phase. Although many are very compelling research challenges, we leave them out of this document in favor of the challenges that arise at the generalization stage and the program adaptation stage. For more details on how we solved the record and replay challenges, we recommend Shaon Barman’s dissertation about Ringer [8].

Challenge 3: Making programs both robust and human-readable

At this point, we shift from discussing the usable drafting tools portion of the work to the learnable languages portion of the work. In particular, given that we must include enough detail about low-level browser internals to make programs robust, how do we also make them readable by humans?

First let us recall why we should care about synthesis languages being learnable. In our case, we are building a DORA workflow, so we need a learnable language in order to handle the ambiguity of single-demonstration specifications.

Even outside of a DORA workflow, expressing synthesized programs in a readable, understandable language is critical for building up users’ trust of the synthesized outputs. If we want users to be comfortable running synthesized programs for high-stakes tasks, we probably want to produce an output that the user can understand and approve.

Producing editable outputs also changes the class of programs end users can access via synthesis. If the synthesized program produces a large output—say three million rows of data—the user cannot inspect the output to approve or reject a synthesized program. If the synthesized program runs for hours or days, they may be able to inspect the output to approve or reject it, but the debugging loop becomes extremely slow and cumbersome. Expressing synthesis outputs in a learnable language lets users identify issues even for programs with large outputs or long execution times.

```

load https://scholar.google.com/citations?hl=en&view... into page1
for each row in authors in page1 (✓ for all rows, for the first 20 rows)
do
  scrape name in page1
  click name in page1, load page into page2
  for each row in papers in page2 (✓ for all rows, for the first 20 rows)
  do
    scrape title in page2
    scrape citations in page2
    scrape year_published in page2
    add dataset row that includes: name TEXT title TEXT citations TEXT

```

Figure 2.17: Our Google Scholar authors-papers program, expressed in the Helena programming language. The Helena synthesizer produces this exact program from a demonstration in which the user demonstrates how to collect the first author in a list of authors, click on the first author, then collect the first paper by the first author. Here we show the program as it is displayed to the end user, in a blocks-based program editor.

To understand this third challenge, we need some additional background on website internals, particularly the JavaScript event loop. When a user does an action they perceive as a click, the webpage sees not a high-level click event, but a sequence of low-level DOM events—*e.g.*, `mousedown`, `mouseup`, `click`, `blur`, `focus`. Likewise, what a user perceives as a unified typing action appears to the webpage as a sequence of `keypress`, `input`, `keydown`, `keyup`, `textInput`, and other events. In the course of even a short interaction, it is common to trigger hundreds of DOM events. Webpage designers can attach handlers to DOM events, which are then called when a user triggers the target event. Automation tools within the browser do not have access to the high-level interface that the human user gets—clicking, typing—but only to the lower-level DOM event interface.

Traditional web record and replay tools handle this issue by mapping each sequence of user-raised DOM event to a high-level summary action, then mapping each high-level summary action down to a standardized sequence of low-level DOM events. At replay-time, they simulate the sequence of low-level DOM events. The problem with this approach is that, as illustrated in Figure 2.16, when users raise a given high-level event, it does not always produce the same sequence of low-level DOM events. Even clicking on the same node on the same webpage can raise different sequences of events depending on the state of the webpage.

Variance in low-level event sequences does not necessarily need to be a problem for web automators. If webpages only ever attached JavaScript handlers to the events that appear every time a given high-level action occurs, it could be enough to map the high-level actions to the guaranteed DOM events. However, modern webpages are highly interactive; they attach handlers to any and all events the user might raise over the course of an interaction. If the program does not raise the same sequence of events as a human user doing the same task, some of a webpage’s critical behavior may simply never be triggered. Recall, for instance, the example from the beginning of this section in which the failure to raise `keypress` events meant that a webpage never populated an autocomplete menu. Thus, to be robust to modern webpages’ high interactivity, Helena must record the entire sequence of low-level DOM events during a recording and simulate them all during a program execution.

```

[event907]type:dom
type:mousedown
xpath:HTML/BODY[1]/DIV[8]/DIV[3]/DIV[5]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[3]/A[1]
URL:https://www.google.com/search?
ei=xtycXdJMuOGggfO4Z3IDg&q=programming+by+demonstration+programming
+by+demonstration&oq=programming+by+demonstration+programming+by+demonstration&gs_l=psy-ab.3..33i299i3.7895.10651..10836..0.2..0.159.2438.25j4.....0....1..gws-wiz.....0i71j0i22i30j33i160j33i22i29i30.ISyglRZuQdw&ved=0ahUKEwjZxKO-rI3IAhVjg-AKHc5wB-kQ4dUDCA&uact=5
port:10
[event908]type:dom
type:focus
xpath:HTML/BODY[1]/DIV[8]/DIV[3]/DIV[5]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[3]/A[1]
URL:https://www.google.com/search?
ei=xtycXdJMuOGggfO4Z3IDg&q=programming+by+demonstration+programming
+by+demonstration&oq=programming+by+demonstration+programming+by+demonstration&gs_l=psy-ab.3..33i299i3.7895.10651..10836..0.2..0.159.2438.25j4.....0....1..gws-wiz.....0i71j0i22i30j33i160j33i22i29i30.ISyglRZuQdw&ved=0ahUKEwjZxKO-rI3IAhVjg-AKHc5wB-kQ4dUDCA&uact=5
port:10
[event909]type:dom
type:mouseup
xpath:HTML/BODY[1]/DIV[8]/DIV[3]/DIV[5]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[3]/A[1]
URL:https://www.google.com/search?
ei=xtycXdJMuOGggfO4Z3IDg&q=programming+by+demonstration+programming
+by+demonstration&oq=programming+by+demonstration+programming+by+demonstration&gs_l=psy-ab.3..33i299i3.7895.10651..10836..0.2..0.159.2438.25j4.....0....1..gws-wiz.....0i71j0i22i30j33i160j33i22i29i30.ISyglRZuQdw&ved=0ahUKEwjZxKO-rI3IAhVjg-AKHc5wB-kQ4dUDCA&uact=5
port:10
[event910]type:dom
type:click
xpath:HTML/BODY[1]/DIV[8]/DIV[3]/DIV[5]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/DIV[3]/A[1]
URL:https://www.google.com/search?
ei=xtycXdJMuOGggfO4Z3IDg&q=programming+by+demonstration+programming
+by+demonstration&oq=programming+by+demonstration+programming+by+demonstration&gs_l=psy-ab.3..33i299i3.7895.10651..10836..0.2..0.159.2438.25j4.....0....1..gws-wiz.....0i71j0i22i30j33i160j33i22i29i30.ISyglRZuQdw&ved=0ahUKEwjZxKO-rI3IAhVjg-AKHc5wB-kQ4dUDCA&uact=5
port:10

```

Figure 2.18: **Low-Level DOM Events Program.** The low-level DOM events required for robust replay are too numerous and too dense for non-programmers to read. These four events together represent one click. The representation in this figure elides most of the detail required to replay these events (and thus the detail required in our replay program), but already it is clear a program with hundreds of these events is unreadable.

Modern, highly interactive pages thus introduce a critical design conflict. We want to show the user a readable program with high-level abstractions, like the program in Figure 2.17, something they can understand and manipulate. However, if we want the programs to run, to complete the intended tasks, they must include hundreds upon hundreds of low-level DOM events, each packed with the recorded event attributes (including hundreds of target node features for deferred node selector synthesis) that let us simulate the event. In short, even a simple straight-line trace of all events logged during a demonstration is long, dense, and difficult to read. (See Figure 2.18 for a simplified but still unreadable DOM event-level representation of one click in a longer program.) Even programmers do not want to read or edit programs in this low-level language, and non-programmers certainly do not.

So should we choose to sacrifice robustness or sacrifice readability? The answer of course is to throw away the old problem formulation, in which we must choose *one* language in which to represent the synthesized program.

Helena introduces a bi-level DSL approach to end-user program synthesis. The synthesizer produces its output program in the low-level DSL, which includes each individual DOM event that the program must raise. Once we have a program in this low-level language, the Helena reverse-compiler translates the program into the high-level Helena language, the language that the

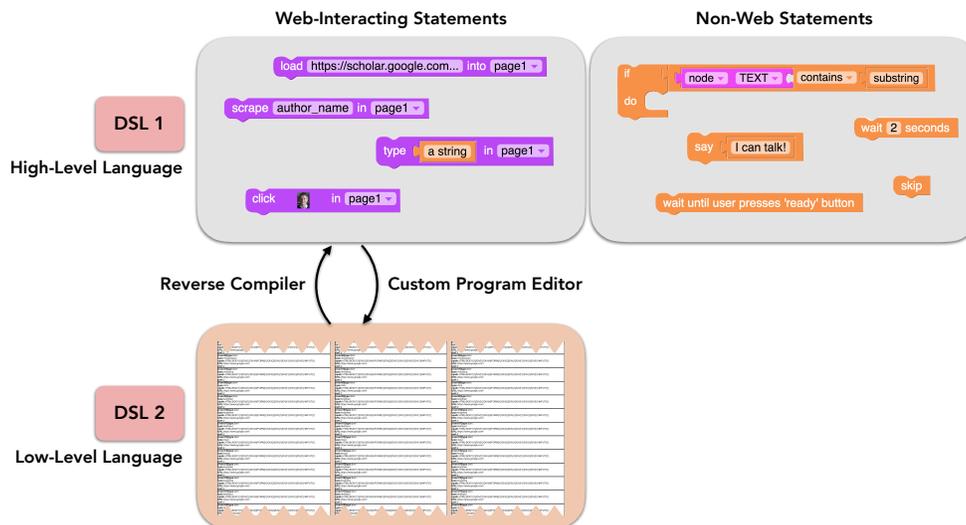


Figure 2.19: **Helena’s Bi-Level DSL.** A bi-level DSL lets us handle the web automation domain even though it is *abstraction resistant*. Programming domains in which writing robust programs requires writing low-level code are abstraction resistant. In the web automation domain, we must replay all low-level DOM events, often hundreds of them, to ensure we trigger all required JavaScript behaviors. However, code that includes all of these details is too low-level for end users to write from scratch and even for them to read and edit. Instead, the Helena synthesizer writes programs in the low-level DSL. Next the Helena reverse compiler maps slices of the low-level programs to statements in a higher-level language. The user can make edits at the higher level. If the edits only involve statements that do not interact with webpages (*e.g.*, control flow), no changes are necessary. If the edits involve statements that do interact with webpages, the changes are mapped back to the low-level language.

human user can read, understand, and edit. Finally, program edits made at the Helena level are mapped back down to the low-level language. Figure 2.19 depicts this bi-level DSL and how reverse compilation links them. Also note that not all statements in the Helena language are mapped to statements in the low-level language—only constructs that involve interactions with webpages need to be linked to the second DSL. Thus, a `click` statement is mapped down, but a `break` statement is not.

Our bi-level DSL lets Helena synthesize large, messy, complicated programs that we cannot expect end users to read, while still making the output programs editable by end users. The outcome is that Helena does handle modern interactive web pages much more effectively than the prior state of the art, as shown in Figure 2.20. The comparison is not strictly fair, because the prior state-of-the-art tool, CoScripter, was developed approximately a decade ago. That was before webpages had become as interactive as they are now, so it is unreasonable to expect CoScripter to handle the interactivity of modern webpages. The better way to look at this result is to recognize that web replay in the face of highly interactive pages is a new problem that had not previously been solved.

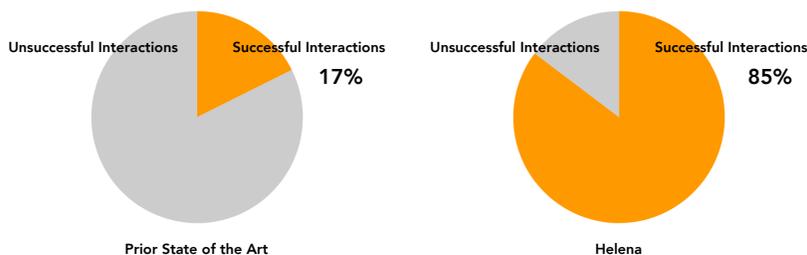


Figure 2.20: Percentage of webpage interactions replayable with CoScripter, the prior state of the art in web record and replay, versus with Helena. Note that neither tool reaches 100% accuracy because some interactions should fail—*e.g.*, bidding on a closed auction.

This bi-level DSL approach lets us handle one of the key challenges of modern end-user programming: *abstraction resistance*. Non-programmers and novice programmers are keen to produce programs for a broad and diverse set of programming domains. The critical hurdle is that many of those domains are abstraction-resistant: they demand that someone writes low-level code. There is no high-level DSL or set of abstractions that can encode all the information required to generate the low-level code via traditional compilation. Thus, we cannot simply wait for someone to produce the high-level library that will make this domains accessible. However, low-level code is exactly the style of code that non-traditional programmers are least prepared to write. Thus, if we want to facilitate end-user programming in abstraction-resistant domains, we must offer new programming techniques that will let them produce low-level code through other means. In the Helena case, our PBD tool writes the low-level code based on user demonstrations and thus puts the abstraction-resistant web automation domain in reach.

The bi-level DSL design comes with some advantages and some disadvantages. Chief among the advantages: the user can tackle domains (like web automation) that simply cannot be abstracted enough to permit high-level DSLs or other high-level abstractions. Chief among the disadvantages: from a tool evaluation perspective, it becomes hard to disentangle the effects of the program synthesis tool and the effects of the language design. Ideally we would like to compare the success rate of users with the language alone against the success rate of users with the language and the synthesizer to identify how much of the gain relative to traditional programming comes from a better language versus how much comes from the synthesizer. However, the bi-level DSL approach means that there is in fact no (reasonable) way to write a Helena program without the synthesizer, so there is no easy way to tease apart the effects of the synthesizer and the language. One key *potential* disadvantage: the high-level DSL is of course less expressive than the low-level DSL and thus offers less control over the final program. Fortunately, this disadvantage can be mitigated with good tool design. With enough escape hatches—ways of letting the user transition from editing the high-level program to editing aspects of the low-level program—the bi-level DSL approach need not sacrifice expressiveness.

Challenge 4: Facilitating end-user parallelization

If one gives a tool like Helena to real end users, it turns out they quickly begin attempting very large tasks, and they start wanting their programs to collect large amounts of data in short amounts of time. Naturally, when this occurred, we started thinking about parallelization. Why wait days for an automation program to run when we could wait mere minutes?

The good news for parallelization: serial Helena web automation programs are not CPU-bound. Their execution times are primarily driven by network latency. This means even parallelizing them on a social scientist's laptop, we can expect to get dramatic performance improvements.

The bad news for parallelization: if we browse the web, reach a particular webpage w , copy its URL u from the browser bar, then paste u into another browser, there is no guarantee that we will get w back. Modern webpages are stateful, interactive, and their contents are updated and shuffled and replaced at a fast rate. Given those characteristics, we can no longer expect the web to behave like a stable map from URLs to DOM trees. The reason this matters for parallelization is that typically in the past, scrapers used URLs to transfer work between parallel workers. At this point in the web's development, this is no longer a reliable general-purpose way to move work between workers.

If we cannot use URLs as pointers to a given subtask, what alternatives do we have? We can always transfer work from the web server itself to individual workers—that is, loading a URL will let a parallel worker access *some* subtask or subtasks, if not a particular fixed subtask. The problem here is that servers are allowed to serve whatever subtasks they want to any given browser, and indeed they do this. Say we want to collect details about the top 1,000 restaurants in our city, according to Yelp. If we set two workers to collect this list at the very same time, we have found that about 11% of the restaurants in one list will not be included in the second list. The lesson here is that web servers serve inconsistent workloads to parallel workers, and any web automation parallelization approach must be able to gracefully handle the fact that each worker may see a dramatically different set of tasks, even if they start from the same URL.

Even for a programmer, this is a rather peculiar sort of parallelization problem, with a totally unknown task size (How many authors will this list include? How many papers will each author have?) and workers pulling subtasks from an unreliable central store (Will workers see the same set of restaurants? In the same order?) that the programmer does not control. If this is confounding for a programmer who already understands the basics of what parallelism means and how to write parallel programs, expecting a non-traditional programmer to sit down and write the fork-join variant of an automation script seems out of reach.

Efforts to discuss parallelization with our social science collaborators largely floundered. The initial goal was to teach them standard parallelization techniques. However, most of them had only faint ideas of what parallelism means in the context of computing, and they were not sufficiently comfortable with traditional programming to use even, for instance, Python parallelization libraries. Fortunately a side conversation revealed that although asking them to parallelize data collection programs only turned up blank looks, asking them to de-duplicate the outputs of those same programs was a smooth process. Data de-duplication was a task they tackled regularly, and it came naturally to them. At that point, we decided to steal the information they would use for de-duplicating scraper outputs and use that same information to transparently parallelize the scrapers.

On the surface, de-duplication and parallelization seem like different tasks, but the key is that both can be boiled down to one goal: to avoid redoing work that has already been done. Determining what work has already been done is complicated, because determining which object snapshots come from the same object is complicated.

Recall that web data changes constantly and that different parallel workers see different data variants. Object attributes change. Are two tweets with the same text but different retweet counts and different comment counts the same object? Are two tweets with different text but the same retweet and comment counts the same object? The answers can change based on the user's goals. If a user wants a set of rental listing addresses, one per rental property, a listing object remains the same even if the price changes. On the other hand, if the user wants to track how each property's listed price changes over time, the same listing with an updated price is a new object. We cannot automatically choose criteria by which to decide if two objects are the same, but we *can* rely on the user to tell us what attributes to consider.

Once we have a way of identifying which objects are the same as previously encountered objects, we have a way of parallelizing. If a program execution encounters an object that has already been claimed—by another parallel worker or because it appears in the dataset multiple times—the program should skip over that object. On the other hand, if it encounters an object that no workers have seen before, the program should complete all work associated with that object. This is the idea behind our novel *skip block* construct.

Below, we use the skip block in the context of our authors-papers working example:

```
for (aRow in p1.authors){
  skipBlock(Author(aRow.author_name, aRow.author_institution)){
    scrape aRow.author_name
    scrape aRow.author_institution
    p2 = click aRow.author_name
    for (pRow in p2.papers){
      scrape pRow.title
      scrape pRow.citations
      output([aRow.author_name, pRow.title, pRow.citations])
    }
  }
}
```

Note the skip block in the outermost loop. The line `skipBlock(Author(aRow.author_name, aRow.author_institution))` indicates that if the current author has the same name and the same institution as a previously encountered author object, we should assume it is the same author, and we should skip over the associated block of code. The body of the skip block is the block of code that operates on author objects, the statements we skip if the author has already been claimed.

From this point, parallelization is straightforward. All parallel workers begin at the beginning of the program. All objects that do not map to the same object based on the skip block's key attributes are considered independent subtasks that can be run in parallel. Helena maintains a claim log. Helena maps the skip block key attributes for each object to a unique ID. When a parallel worker encounters a new object, it checks if the UID has been claimed. If yes, it skips the block and proceeds to the next object. If no, it executes the block.

Note that we can also use hashing instead of a claim log. The key is to use the attribute-based UIDs to distribute the independent subtasks. Interestingly, data churn is so common that using a

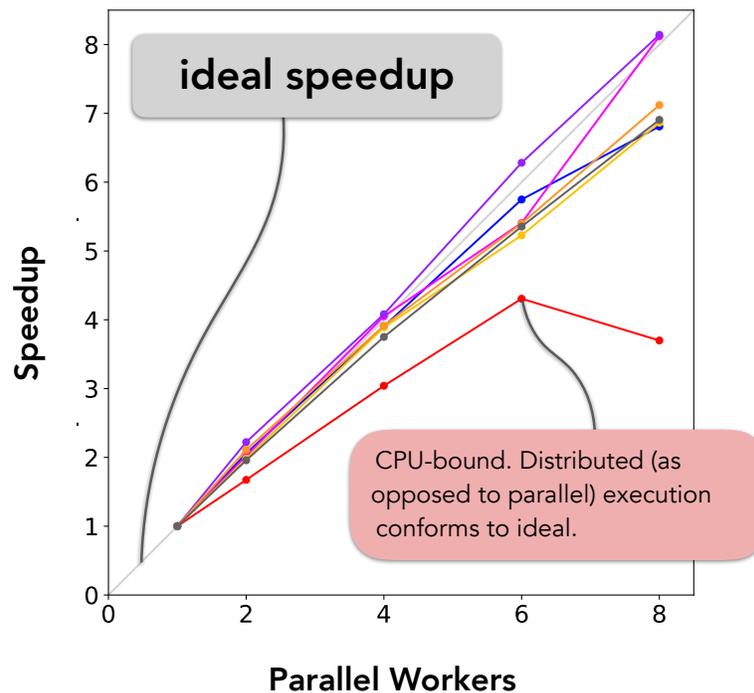


Figure 2.21: **Skip Block Parallelization Performance.** Speedup from parallelization on a single machine. For most of the automation tasks, the speedups hew close to the ideal. One task (here in red) finally becomes CPU-bound; this task uses webpages that are very large not in terms of inches but in terms of number of nodes. Distributing the task over multiple machines (with skip blocks, distributing execution is just as easy as parallelizing) brings this task up to the ideal speedup.

hash-based approach instead of a claim log actually alters what data appears in the final output dataset.

Ultimately, the key to putting parallelization, a very complicated programming task, in range for non-traditional programmers was a problem reframing. Rather than asking for something that is difficult for them, to parallelize a serial program, Helena asks for something that comes very naturally to them, a de-duplication strategy.

Figure [2.21](#) shows the speedups skip blocks achieve on a single machine, quite close to the ideal speedup. With distributed execution—equally easy with the skip block—speedups are even closer to the ideal. In early experiments with interactive-time scraping, we have observed 70x speedups, bringing hour-long tasks down under the one-minute mark. These results are promising, given that the Helena runtime was designed for long-running scraping tasks and has not yet been optimized for fast startup times.

We can also use skip blocks for a variety of other purposes, beyond parallelization and distribution. For instance, failure recovery turns out to be very important for our target audience. An early stumbling block in one of our collaborations came when a research team started doing longitudinal studies using a Helena scraper. The undergraduate in charge of running the program had it run every

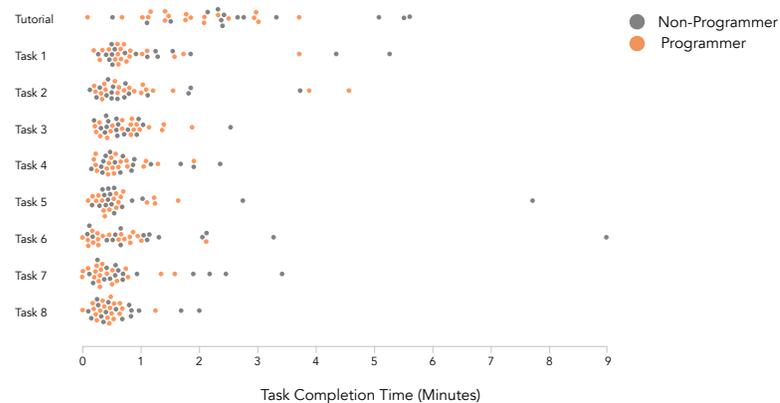


Figure 2.22: **Skip Block Ease of Use.** Orange dots represent programmers, and gray dots represent non-programmers. Non-programmers add a new skip block—parallelize a new program—in about one minute, non-programmers in a little less than one minute.

night on his laptop, on his home WiFi network. At one stage, the WiFi was cutting out every night, interrupting the executions, and when he would wake up in the morning and restore the connection, the program would repeat all the same work it had done the night before. With the skip block, the same program can essentially fastforward over work that has already been done, so the skip block offers simple, graceful failure recovery. Skip block recovery runs achieve 7% overhead relative to hypothetical zero-time recovery strategies (which do not even exist for many automation tasks).

Users can also deploy skip blocks for incrementalization. By default, a skip block skips over an object if it has ever been committed to the commit log or claimed in the claim log. Without going into the details of the skip block syntax, the user does have the option to adjust this behavior. For instance, we can use skip blocks to direct a program to rescrape any author who has not been scraped in the last year, or in the last three scrapes, which is a convenient way to make sure we eventually update data in longitudinal studies. Running the whole program for each run of a longitudinal study could be very slow. We might prefer to incrementalize. We observe speedups of $1.8\times$ to $800\times$ for real incrementalized Helena programs.

Most crucially, the skip block is easy for people to use. Within the Helena programming environment, it is a simple matter of checking the boxes for the key attributes, and then the tool can add a corresponding skip block to the user’s program. User study results make it clear the skip block is learnable and usable. See Figure [2.22](#). Non-programmers take only about 7 minutes and programmers only about 2 minutes to learn how they work. After that, non-programmers can parallelize each new program in about 61 seconds and programmers in about 52 seconds. Since producing a language construct that makes parallelization a one-minute task is a stretch goal for *any* audience, the fact that skip blocks turn parallelization into a one-minute task for people who consider themselves non-coders is a very encouraging result.

Chapter 3

Programming Model

Programming by Demonstration (PBD) promises to enable data scientists to collect web data. However, in formative interviews with social scientists, we learned that current PBD tools are insufficient for many real-world web scraping tasks. The missing piece is the capability to collect hierarchically-structured data from across many different webpages. We present the Helena synthesis tool, a programming system for writing complex web automation scripts by demonstration. Users demonstrate how to collect the first row of a ‘universal table’ view of a hierarchical dataset to teach Helena how to collect all rows. To offer this new demonstration model, we developed novel relation selection and generalization algorithms. In a within-subject user study on 15 computer scientists, users can write hierarchical web scrapers 8 times more quickly with Helena than with traditional programming.

3.1 Introduction

Web data is becoming increasingly important for data scientists [89, 56]. Social scientists in particular envision a wide range of applications driven by web data:

“**forecasting** (e.g., of unemployment, consumption goods, tourism, festival winners and the like), **nowcasting** (obtaining relevant information much earlier than through traditional data collection techniques), **detecting health issues** and well-being (e.g. flu, malaise and ill-being during economic crises), **documenting the matching process** in various parts of individual life (e.g. jobs, partnership, shopping), and **measuring complex processes** where traditional data have known deficits (e.g. international migration, collective bargaining agreements in developing countries).” [6]

To use web datasets, data scientists must first develop web scraping programs to collect them. We conducted formative interviews with five teams of data scientists to identify design requirements for web data collection tools. The first critical requirement: do not require knowledge of HTML, DOM trees, DOM events, JavaScript, or server programming. Our formative interviews revealed that when data scientists attempt to use traditional web scraping libraries—e.g., Selenium [83],

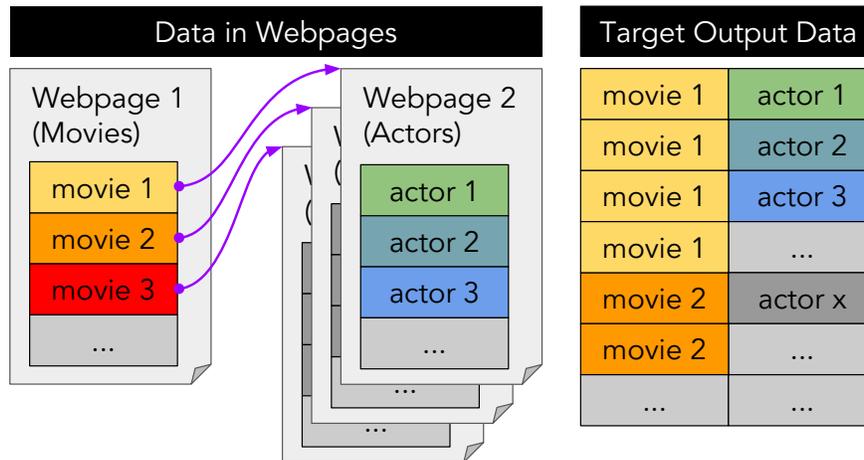


Figure 3.1: An example of a distributed, hierarchical web dataset. The goal is to collect a list of movies and, for each movie, a list of actors. The placement of data cells in webpages appears at left; one webpage lists multiple movies, and each movie links to a details webpage, which lists the movie’s cast. The target output data appears at right; each row includes a movie and an actor who acted in the movie. The data is distributed because it appears across multiple pages: one page that lists movies and a set of movie details pages. The data is hierarchical because the root is a parent of multiple movies and each movie is a parent of multiple actors. In our formative study, 100% of data scientists’ target web datasets were distributed, and 50% were hierarchical.

Scrapy [80], BeautifulSoup [73]—they often find themselves lacking the requisite browser expertise, especially when they need to reverse engineer browser-server communication.

Programming by Demonstration (PBD) delivers on this first design requirement, offering web automation without requiring users to understand browser internals or manually reverse engineer target pages. The PBD approach has produced great successes in the web automation domain, most notably CoScripter [51], Vegemite [53], and iMacros [33], but also others [54, 82, 37, 32, 50]. CoScripter and iMacros offer record-and-replay functionality; users record themselves interacting with the browser—clicking, entering text, navigating between pages—and the tool writes a loop-free script that replays the recorded interaction. Vegemite adds a *relation extractor*, a tool for extracting tabular data from a single webpage into a spreadsheet. By putting its relation extractor and the CoScripter replayer together, Vegemite lets users extend a data table with new columns; users can invoke a loop-free CoScripter script on each row of an extracted table, using the row cells as arguments to the script and adding the return value as a new cell. The end result is a PBD system that can collect tables of data even if cells come from multiple pages.

Our formative interviews revealed several key design requirements, but our interviewees emphasized that one in particular is critical to making a scraping tool useful and has not yet been met by prior PBD tools: collecting realistic, large-scale datasets. In particular, scrapers must handle **distributed data**, i.e., data that is dispersed across multiple webpages, and they must handle **hierarchical data**, i.e., tree-structured data.

Distributed Data

The data scientists in our formative interviews indicated that they care about *distributed data*—that is, datasets in which the data that constitutes a single logical dataset is spread across many physical webpages. For instance, to scrape information about all movies in a box office top-10 list, we may scrape the title of each movie from the list page, then follow a link to the movie’s details page to scrape the name of the movie’s director, then follow a link to the director’s profile page to scrape the director’s bio. In this case, each of the three columns (movie title, director name, director bio) appears on a different page.

In general, web scraping includes two subtasks:

- *Single-page data extraction*: Given a page, collecting structured data from it; e.g., finding an element with a given semantic role (a title, an address), extracting a table of data.
- *Data access*: Reaching pages from which data should be extracted, either by loading new pages or causing new data to be displayed in a given page; e.g., loading a URL, clicking a link, filling and submitting a form, using a calendar widget, autocomplete widget, or other interactive component.

Many PBD scraping tools only write single-page extraction programs (e.g., Sifter [31], Solvent [58], Marmite [90], FlashExtract [50], Kimono [37], import.io [32]). A few PBD scraping tools write programs that also automate data access; in particular, record and replay tools (e.g., Ringer [9], CoScripter [51], iMacros [33]) and the Vegemite mashup tool [53] can write programs that do both extraction and data access.

Because distributed datasets split logically connected data points across many pages, a PBD tool that aims to scrape realistic data must synthesize scripts that automate data access.

Hierarchical Data

Our formative interviews also revealed that data scientists want to collect hierarchical web datasets. See Fig. 3.1 for an example of a hierarchical dataset. The task is to scrape, starting from a list of movies, the title of each movie, then for all actors in each movie, the name of each actor. (Note that this data is also distributed; the list of actors for each movie appears on a movie-specific page, not on the page that lists all movies.) To scrape this data, we need a program with nested loops: an outer loop to iterate through movies and an inner loop to iterate through actors for each movie.

To scrape distributed data, we need to automate data access. To scrape hierarchical data, we need nested loops. However, to date, PBD scrapers that support data access all synthesize scripts in languages that lack loops. The Ringer [9] language, iMacros language [33], and CoScripter language [51] all lack loops. Vegemite [53] uses the CoScripter language but comes closer to offering loops; by offering a UI that allows users to ‘select’ any subset of rows in a spreadsheet and run a loop-free CoScripter script on the selected rows, Vegemite can execute one loop, even though the output program presented to the user is loop-free. However, this approach does not extend to nested loops. One could use Vegemite to extract an initial table of movies, then run a

script to extend each movie row with the top-billed actor, but could not produce the full dataset of (movie, actor) pairs depicted in Fig. 3.1; the script itself cannot extract or loop over tables. These tools not only synthesize programs that lack nested loops but also—because they use loop-free languages—prevent users from adding loops to the straight-line programs they output.

Introducing nested loops is a core outstanding problem in PBD. Even for domains outside of scraping, most PBD systems cannot produce them. Some, like Vegemite, lack algorithmic support (e.g., Eager [17]). Others use algorithms that could add nested loops but prevent it because it makes synthesis slow (e.g., FlashFill [24]).

PBD systems that *can* produce programs with nested loops typically require users to identify loop boundaries (e.g., SMARTedit [47]) or even whole program structures (e.g., Sketch and other template approaches [84]). This is difficult and error-prone. In the SMARTedit study, only 1 of 6 participants could add a nested loop by identifying boundaries, even though the participants were CS majors [48]. Designing interactions that enable automatic nested loop synthesis is an open problem.

To produce scraping programs with nested loops via PBD requires innovations in:

- *Algorithm design*: A synthesis algorithm that can write programs with nested loops based on a demonstration.
- *Interaction design*: An interaction model that produces demonstrations amenable to the synthesis algorithm.

This chapter presents contributions in both algorithm and interaction design. In particular, we introduce a novel interaction model that makes the synthesis problem tractable. In our interaction model, a user demonstrates one row of the ‘join’ of their target output tables. For the Fig. 3.1 movie and actor data, this is the title of the first movie and name of the first actor in the first movie—the first row in the table at the right. Say we want to scrape a list of cities, for each city a list of restaurants, for each restaurant a list of reviews; we demonstrate how to collect the first city’s name, the name of the first restaurant in the first city, then the first review of that first restaurant.

Given a demonstration of how to collect a row of the joined data, our custom synthesizer produces a scraping program with one loop for each level of the data hierarchy. A Relation Selector algorithm leverages common web design patterns to find relations (e.g., movies, actors, cities, restaurants, reviews). A Generalizer algorithm produces a loop for iterating over each relation. This lets our approach introduce arbitrarily nested loops without user intervention. Although our Relation Selector is a web-specific solution, designed to take advantage of web design patterns, our interaction model and Generalizer are not specialized for the web. *The success of our approach for the web scraping domain suggests a general strategy: ask the user to demonstrate one iteration of each nested loop (in our case, collect one row of the joined output table) and use domain-specific insights to identify objects that should be handled together (in our case, the rows of a given relation).*

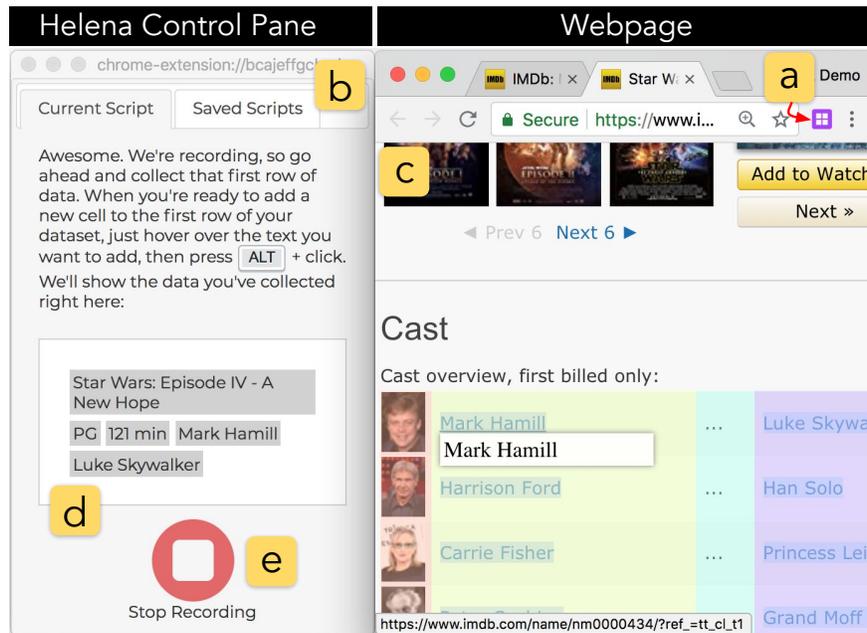


Figure 3.2: Using Helena. The user starts by clicking on (a), the icon for the Helena Chrome extension, which opens (b), the control pane at left. In (c), the normal browser window, the user demonstrates how to interact with pages, collect data from pages, and navigate between pages. The user collects the first row of the target dataset, which appears in the preview pane, (d). When the user clicks on (e), the ‘Stop Recording’ button, Helena synthesizes a script that collects the full dataset.

The Helena Synthesizer

This chapter presents the design the Helena PBD tool, which uses a novel interaction model and novel synthesis algorithms to collect distributed hierarchical data from the web. Helena produces a web scraping program from a single user demonstration. It is the first PBD tool that can collect hierarchical data from a tree of linked webpages.

Fig. 3.2 shows a usage scenario:

- (a) The user opens the Helena browser extension.
- (b) The user starts a demonstration and is asked to collect the first row of the target dataset.
- (c) In the browser window, the user demonstrates how to collect data from pages, interact with page UX elements, and navigate between pages. In this example, the user loads a webpage with a list of movies, collects the title, rating, and length of the first movie, clicks on the movie title to load a movie-specific webpage, then collects the name and role of the first actor in the movie.
- (d) As the user scrapes new data cells, the cells are added to the preview box in the control panel.
- (e) The user ends the demonstration by clicking on the ‘Stop Recording’ button. Helena uses the demonstration of how to scrape the first movie and the first actor of the first movie to (i) detect

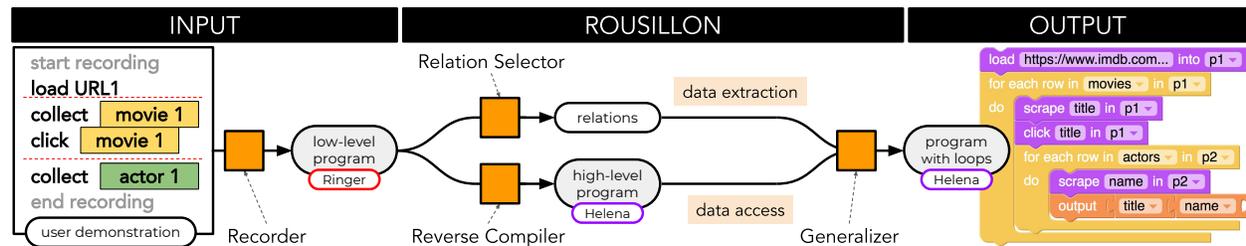


Figure 3.3: The Helena workflow. A user provides a single input demonstration, recording how to collect the first row of the target dataset. For instance, to collect the dataset depicted in Fig. 3.1, the user navigates to a URL that loads a list of movies, collects the first movie’s title, clicks the first movie’s title to load the movie’s details page, collects the name of the first actor in the first movie, then ends the recording. A Recorder converts the user’s demonstration into a program that replays the demonstrated interaction; it collects the first row. This replay program is the input to our PBD tool. Our Relation Selector uses the interacted elements (e.g., movie name, actor name) to identify relations on the target webpages; here, it finds a relation of movies on page one and actors on page two. The input replay program uses the Ringer [9] language, which is low-level and unreadable, so our Reverse Compiler translates from Ringer to our readable, high-level Helena web language. Finally, our Generalizer uses the selected relations (e.g., movies, actors) and the straight-line Helena program to write a Helena program that collects not only the first row of the data but all rows of the data.

relations (movies, actors) and (ii) synthesize a scraping script that iterates through these two relations using the demonstrated interactions. After this process, the user can inspect and edit a blocks-based visual programming language representation of the synthesized script in the control panel.

The Helena architecture is depicted in Fig. 3.3. The input is a single user demonstration, the recording of how to collect the first row of ‘joined’ data. From the trace of DOM events triggered by the user’s interactions, a web record-and-replay tool, Ringer [9], produces a straight-line replay script in the low-level Ringer language. From this single input, Helena extracts the inputs it needs for synthesizing single-page data extraction code *and* data access code. From information about the webpage elements with which the user interacted, Helena’s **Relation Selector** identifies relations over which the user may want to iterate. Helena’s **Reverse Compiler** translates the loop-free Ringer program into a loop-free program into our custom Helena web automation language, a more readable language. Finally, Helena’s **Generalizer** combines the straight-line Helena program with the relations identified by the Relation Selector and produces a Helena program with loops over the relations. This chapter describes the interaction model for producing the input demonstration and the algorithms that drive Helena’s Relation Selector, Reverse Compiler, and Generalizer.

Contributions

This chapter presents the following contributions:

- A set of **design requirements for end-user web scraping tools**, discovered via formative

interviews with five teams of data scientists.

- For the domain of data-access web scraping, **the first PBD tool that synthesizes programs with loops** and the first PBD tool that collects distributed hierarchical data.
- A **user study** demonstrating that programmers can write scraping programs for distributed hierarchical data 8x more quickly with PBD than with traditional programming.

3.2 Related Work

We discuss related work from a broad space of web automation tools, with a special emphasis on tools that use PBD.

PBD Single-Page Data Extraction

PBD tools for data extraction typically ask users to label DOM nodes of one or more sample webpages, then produce extraction functions that take a webpage as input and produce DOM nodes as output. For instance, a *web relation extractor* takes as input a page and a subset of the page’s DOM nodes that constitute part of a logical table, labeled with their row and column indexes. The output is a function that extracts a table of DOM nodes from the labeled page. Vegemite’s *VegeTable* [53] and many industrial tools— e.g., *FlashExtract* [50], *Kimono* [37], and *import.io* [32]—offer PBD relation extraction.

Other PBD single-page extractors collect non-relational data. For instance, to extract product data, a user might label the product name and product price on one or more product pages. The synthesized output is a function that takes one product page as input and produces a name and price as output. Many tools offer this functionality, from the familiar *Sifter* [31], *Solvent* [58], and *Marmite* [90] tools to a vast body of work from the wrapper induction community [19, 13, 44, 91, 42, 66, 20, 3, 43, 61, 29].

Alone, extractors cannot collect distributed data because they cannot synthesize data access. The synthesized programs do not load pages; they take one already-downloaded page as input. By definition, distributed data includes data from multiple pages, and finding and accessing those pages is part of the scraping task. Thus, while data extraction programs may be used within a distributed scraper, they cannot automate the entire collection process.

PBD Data Extraction + Data Access

Most PBD data access tools have focused on *record and replay*. A web record and replay tool or *replayer* takes as input a recording of a browser interaction and produces as output a script for replaying that same interaction. *Ringer* [9], *Selenium Record and Playback* [82], *CoScripter* [51], and *iMacros* [33] all provide this functionality, as do many others [39, 54, 30, 52, 69, 22], although some require debugging by a DOM expert and thus arguably belong outside of the PBD category.

Traditional PBD data access tools focus strictly on replaying the same loop-free interaction and thus cannot be used for large-scale web automation. The exception is the seminal Vegemite tool [53], described in the introduction. Because it can collect distributed data and generalize beyond pure replay, Vegemite comes closest of all existing PBD scraping tools to meeting the needs identified in our formative study. The key differences between Vegemite and Helena are that Vegemite:

- **cannot add nested loops.** See the introduction for a discussion of why it cannot add them, why adding them is a key technical challenge, why they are critical to scraping hierarchical data, and why hierarchical data is desirable.
- **uses a divided interaction model**, requiring one demonstration for data extraction (relation finding) and a separate demonstration for data access (navigation). Users reported “it was confusing to use one technique to create the initial table, and another technique to add information to a new column” [53]; early versions of Helena used a divided interaction model and received similar feedback. Thus, Helena accepts a single demonstration as input and extracts both data extraction and data access information from this one input.
- **does not resolve the robustness-readability tradeoff.** Vegemite uses a less robust replayer, CoScripter, which was designed for an earlier, less interactive web. CoScripter’s high-level language makes its programs readable but fragile in the face of page redesigns and AJAX-heavy interactive pages. On a suite of modern replay benchmarks, Helena’s Ringer replayer successfully replays 4x more interactions than CoScripter [9]. To use Vegemite on today’s web, we would need to reimplement it with a modern replayer that uses low-level statements for robustness. This would leave Vegemite with the same robustness-readability tradeoff that Helena faces (and which drives one of Helena’s critical contributions, its reverse compiler and bi-level DSL).
- **can replace uses of typed strings only.** For example, Vegemite can turn a script that types “movie 1” in *node* into a script that types “movie 2” in *node*. In contrast, Helena can replace uses of typed strings, URLs, and DOM nodes (e.g., click on *node*₂ instead of *node*₁). The browser implementation details that make DOM node replacement more challenging than string replacement are interesting but not related to the key contributions in this chapter; so although this substantially limits the possible applications of Vegemite, we will not emphasize this last distinction.

Another PBD data access approach uses site-provided APIs rather than webpage extraction [14]. This is a good approach for cases in which the website offers an API that includes the target data. However, this is rare in practice; none of the 10 datasets described in our formative study are available via API. (Only one dataset used a site that offers an API, and that site limits the amount of API-retrievable data to less than the team wanted and less than its webpages offer).

Web Automation Languages

There are many Domain Specific Languages (DSLs) for scraping, most implemented as libraries for general-purpose languages: Selenium [83] for C#, Groovy, Java, Perl, PHP, Python, Ruby, and Scala; BeautifulSoup [73] and Scrapy [80] for Python; Nokogiri [63] and Hpricot [28] for Ruby; HXT [26] for Haskell. Some drive a browser instance and emphasize human-like actions like clicks and keypresses; others emphasize methods for parsing downloaded DOM trees, offer no mechanisms for human-like interaction, and thus require users to reverse engineer any relevant AJAX interactions (e.g., BeautifulSoup, Scrapy). To use any of these DSLs, programmers must understand DOM trees and how to traverse them—e.g., XPath or CSS selectors—and other browser internals.

Partial PBD

While the traditional web automation DSLs described above do not use PBD, a class of GUI-wrapped DSLs do mix PBD with traditional programming. Mozenda [60] and ParseHub [67] are the best known in this class, but it also includes tools like Portia [79], Octoparse [65], and Kantu [2]. With these hybrid tools, users build a program statement-by-statement, as in traditional programming, but they add statements via GUI menus and buttons, rather than with a text editor or mainstream structure editor. The user selects the necessary control flow constructs and other statements at each program point, as in traditional programming. However, users write node extraction code via PBD. When they reach a point in the program at which they want to use a node or table of nodes, they use the GUI to indicate that they will click on examples, and the tool writes a function for finding the relevant node or nodes. This class of tools occupies an unusual space because users need to reason about the structure of the program, the statements they will use—essentially they need to do traditional programming—but because these tools' GUIs support such small languages of actions, they do not offer the highly flexible programming models of traditional DSLs.

Helena Building Blocks

Helena makes use of two key building blocks, Ringer [9] and the Helena language. **Ringer** is a web replayer. The input is a user interaction with the Chrome browser, and the output is a loop-free program in the Ringer programming language that automates the same interaction. Helena uses Ringer to record user demonstrations; the Ringer output program is the input to the Helena synthesizer. **Helena** is a high-level web automation language. With statements like `load`, `click`, and `type`, it emphasizes human-like interaction with webpages. The Helena synthesis tool expresses its output programs in the Helena language.

3.3 Formative Interviews and Design Goals

To explore whether current languages and tools for web automation meet data scientists' needs, we conducted formative interviews with five teams of researchers at a large U.S. university, all

actively seeking web data at the time of the interviews. The teams come from a variety of disciplines. One team is comprised primarily of sociologists with two collaborators from the Department of Real Estate. All other teams were single-field teams from the following departments: Public Policy, Economics, Transportation Engineering, and Political Science. We group data collection approaches into three broad strategies: (i) automatic collection with hand-written programs, (ii) manual collection, and (iii) automatic collection with PBD-written programs. The primary focus of each interview was to explore whether a team could meet its current data needs with each strategy.

All teams considered hand-written web scraping programs out of reach, despite the fact that four of five teams had at least one team member with substantial programming experience. The Political Science team even included a programmer with web scraping experience; he had previously collected a large dataset of politicians' party affiliations using Python and Beautiful Soup [73]. He had attempted to collect the team's new target dataset with the same tools, but found his new target website made extensive use of AJAX requests, to the point where he could not reverse engineer the webpage-server communication to retrieve the target data. More commonly we saw the case where one or more team members knew how to program for a non-scraping domain—e.g. visualization, data analysis—but had attempted scraper programming without success because they lacked familiarity with DOM and browser internals. Team members found traditional web automation scripts not only unwritable but also unreadable.

In contrast, two teams considered manual data collection a viable strategy. One team went so far as to hire a high school intern to collect their data by sitting in front of the browser and copying and pasting text from webpages into a spreadsheet. Because their dataset was relatively small—only about 500 rows—this was manageable, albeit slow. Another team scaled down their target dataset to make it small enough to be collected by hand once a week, but the resultant dataset was much smaller than what they initially wanted, and the collection process still took hours of tedious copying and pasting every week. For all other teams, the target dataset was so large that they did not consider manual collection feasible.

For PBD tools, the verdict was mixed. On the one hand, all teams included at least one proficient browser user who felt comfortable demonstrating how to collect a few rows of the team's target datasets, so all teams had the skills to use them. On the other hand, most of the target collection tasks could not be expressed in the programming models of existing PBD web automation tools.

Each team had defined between one and three target datasets. Between them, the five teams had defined 10. All were distributed datasets, and all required scraping at least 500 webpages. Of the 10 target datasets, only two could be collected with existing PBD web automation tools. With a hypothetical variation on Vegemite that extends it (i) from parameterizing only typed strings to parameterizing target DOM nodes and (ii) from handling only stable input relations to handling live relation extraction across many webpages, we might collect as many as three more datasets. However, even this hypothetical improved tool would still fail to express a full half of the target datasets, because five of the 10 datasets are hierarchical.

Based on the challenges revealed in our interviews and the data scientists' stated preferences about programming style, we formulated the following design goals:

- **D1 Expertise:** Do not require knowledge of HTML, DOM trees, DOM events, JavaScript, or

server programming.

- **D2 Distributed Hierarchical Data:** Handle realistic datasets. In particular, collect hierarchical data, including hierarchical data that can only be accessed via arbitrary webpage interactions and navigating between pages.
- **D3 Learnability:** Prioritize learnability by tool novices over usability by tool experts.

Guided by these goals, we designed the Helena PBD web scraper, which can collect all 10 of the web datasets targeted by the teams in our formative study.

3.4 Helena Interaction Model

To synthesize a scraper for distributed, hierarchical data, we need to acquire the following information from the user:

1. how to interact with and navigate to pages (data access)
2. all relations over which to iterate (data extraction)
3. the links between relations, the structure of the hierarchy

Whatever input a PBD tool takes, it must solicit all of these.

Single-Demonstration Model for Learnability. One possible approach is to require users to complete a different type of demonstration for each of the three information types. However, users find it hard to learn divided interaction models in this domain [53]. A multi-demonstration approach thus inhibits design goal D3 (prioritizing learnability). To offer a learnable interaction, PBD scrapers should aim to use only one demonstration.

Helena makes a contract with the user, restricting the form of the demonstrations they give, and in exchange requesting only one demonstration. In particular, we designed Helena to accept demonstrations in which users collect the first row of all relations in the target output data; essentially, users must demonstrate the first iteration of each loop that should appear in the final scraping program. A demonstration of this form has the potential to offer all of the requisite information types: (1) to reach the first row data, the user demonstrates all interaction and navigation actions; (2) the user touches all relations of interest; and (3) the order in which the user touches relations implicitly reveals how they are linked.

With the form of the input demonstration fixed, a few key interaction design problems arise: communicating what data users can scrape from a given element, communicating what loops users can add, and communicating how Helena’s scraping program will operate.

Gulf of Execution. To use a PBD scraper well, users must know what data they can collect and what loops they can add. What data can be scraped from a photograph? From a canvas element? If a user interacts with the title and year of one movie in a list, will the scraper know to generalize to interacting with the titles and years of all movies? To reduce the *gulf of execution* [64], a PBD scraper should help users answer these questions about how to use the system.

Gulf of Evaluation. Users also benefit from understanding the program a PBD scraper is building. In particular, based on the demonstration provided so far, what kind of output data will the scraper produce? Short of completing the demonstration and running the output program, how can users learn if the program will collect the data they want? To reduce the *gulf of evaluation* [64], a PBD scraper should help users answer this question about the current state of the system.

Interface

Here we describe the interface of the Helena PBD scraping tool and how it addresses key interaction design problems.

Implementation. Helena is a Chrome extension, available at <http://helena-lang.org/download> and open sourced at <https://github.com/schasins/helena>. We designed Helena as a Chrome extension to make installation easy and fast. Chrome extension installation is a drag-and-drop interaction, so this design choice reduces the setup and configuration complexity relative to classical web automation tools like Selenium.

Activation. The user activates the Helena browser extension by clicking on its icon in the URL bar (see Fig. 3.2(a)). Upon activation, Helena opens a control panel from which the user can start a new recording, load saved programs, or download data from past program runs. Because the focus of this chapter is on using PBD to draft scrapers, we discuss only the recording interaction for writing new programs.

Recording. Users begin by clicking a ‘Start Recording’ button. This opens a fresh browser window in which all the user’s webpage interactions are recorded. Users are instructed to collect the cells of the first row of their target dataset.

Scraping. Users can scrape data by holding the ALT key and clicking on the webpage element they want to scrape (see Fig. 3.4). This indicates to Helena that the element’s content should be the next cell in the first row of the output dataset.

Gulf of Execution. For the most part, webpages look the same during recording as in everyday browsing. However, we make two modifications to help users discover what Helena can do with their recorded actions, to bridge the gulf of execution.

- **What can we scrape?** Hovering over any element in a webpage displays an in-site output preview beneath the element, a hypothetical cell (see Fig. 3.4). This previewed cell communicates the content that Helena will extract if the user decides to scrape the element. The user learns what scraping an element would mean without having to write and then run a program that scrapes it. Previews are especially useful for (i) elements with text content that comes from `alt` and `title` attributes rather than displayed text and (ii) non-text elements (e.g. images are non-text elements, so Helena collects image source URLs).
- **What loops can we add?** Hovering over any element that appears in a known relation highlights all elements of the known relation (see Fig. 5). This emphasis of relation structure communicates that Helena knows how to repeat interactions performed on a row of the relation for all rows of the relation; it gives users additional control over the PBD process

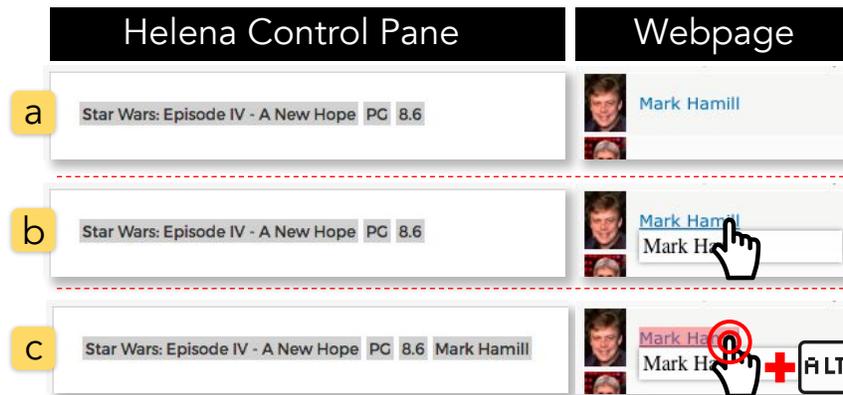


Figure 3.4: Scraping data. Boxes on the left show snippets of the Helena control pane at various points during demonstration. Boxes on the right show snippets of the webpage with which the user is interacting, at the same points in time. a) The user has already added a movie title, PG rating, and star rating to the first row of data, shown in the first row preview at left. In the webpage, we see an actor’s picture and link, but the user is not hovering over them. b) An in-site output preview. When the user hovers over an element, an inline preview shows a cell the user could add to the first row. Here the user hovers over the actor name. c) The user holds the ALT key and clicks on the actor name. The text from the preview cell is added to the first row of data (see preview at left).

by indicating they can interact with a given set of elements to add a given loop in the output program.

Gulf of Evaluation. When the user scrapes a new element, Helena adds its content to a preview of the first row, displayed in the control pane (see Fig. 3.4). This preview of the output data reduces the gulf of evaluation by giving the user information about the program Helena is building. Helena’s output program will always produce the previewed row of data as its first output row, as long as the structure and content of the webpages remain stable. If at any point the user realizes that the preview does not show the intended data, they can identify without running the program—without even finishing the demonstration—that the current demonstration will not produce the desired program.

Finishing. When the user ends a demonstration by clicking the ‘Stop Recording’ button (see Fig. 3.2(e)), the recorded interaction is passed as input to Helena’s PBD system, described in the Algorithms section. The output of the PBD system is a program in the high-level Helena web automation language; the control pane displays this program in a Scratch-style [72] blocks-based editor (see output in Fig. 3.3). At this stage, the user can edit, run, and save the new program.

3.5 Algorithms

In this section, we describe three key Helena components: the Reverse Compiler, Relation Selector, and Generalizer. Fig. 3.3 shows how these components interact to write a scraping program based on an input demonstration.

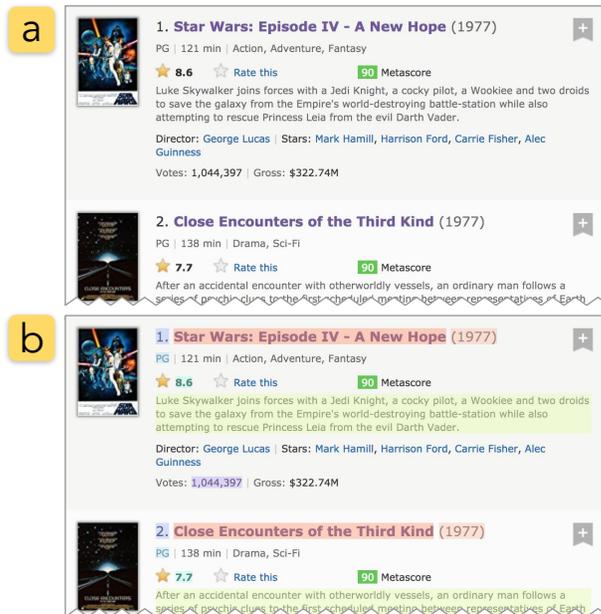
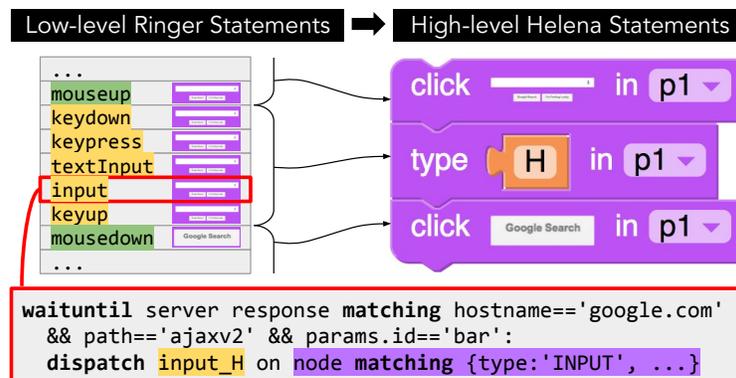


Figure 3.5: A centralized server stores relations used in prior Helena programs; Helena uses stored relations to communicate about loops it can add. When a user hovers over a known relation during demonstration, Helena highlights the relation to indicate it knows how to loop over it. Here, (a) shows an IMDb.com page, 1977’s top box office movies, and (b) shows the same page with the movies relation highlighted.

Reverse Compiler



Problem Statement: Traditional replayers write high-level, readable scripts but are fragile on today’s interactive, AJAX-heavy webpages; modern replayers work on interactive pages but write low-level, unreadable programs.

Our Solution: Use a modern replayer for robustness, but reverse-compile to the Helena language to recover readability.

Helena uses Ringer to record user demonstrations (Recorder in Fig. 3.3). We chose Ringer because it outperforms alternatives on robustness to page content and design changes [9]. However, the output of a Ringer recording is a script in Ringer’s low-level language, with statements like the red-outlined statement in the Reverse Compiler figure. With one statement for each DOM event—e.g., `keydown`, `mouseup`—a typical Ringer script has hundreds of statements, each tracking more than 300 DOM node attributes. This exhaustive tracking of low-level details makes Ringer

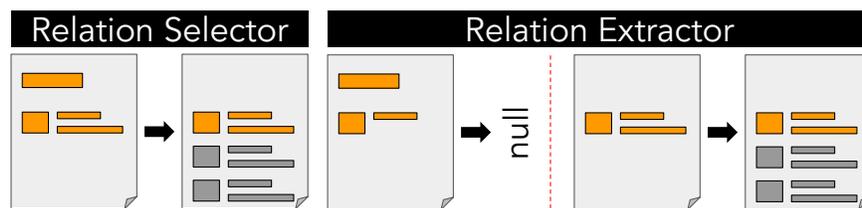
scripts robust, but it also makes them unreadable even for DOM experts. To meet Design Goal D1, sheltering users from browser internals, we hide these details.

The Reverse Compiler translates Ringer programs into our high-level web automation language. It produces high-level statements that a user can read, but whose semantics are determined by associated snippets of code in the Ringer language—the low-level statements that make replay robust.

The Reverse Compiler slices the input Ringer program into sequences of consecutive statements that operate on the same target DOM node and are part of one high-level action; then it maps each sequence of Ringer statements to a single Helena statement. Each Ringer statement includes the type, t , of its DOM event and the DOM node, n , on which it dispatches the event. Browsers use a fixed set of DOM event types, T , and Helena has a fixed set of statements, H . The Helena synthesizer uses a map $m : T \mapsto H$ that associates each type in T with a high-level Helena statement; for example, `keydown`, `keypress`, `textInput`, `input`, and `keyup` all map to Helena's `type` statement. The synthesizer uses m to label each Ringer statement: $(t, n) \rightarrow (t, n, m(t))$. Next it slices the Ringer program into sequences of consecutive statements that share the same n and $m(t)$ ¹. Because many event types are mapped to a single Helena statement type, the Ringer sequences are typically long. Next, the Reverse Compiler writes the Helena program; it maps each sequence of Ringer statements to a single Helena statement that summarizes the effects of the whole Ringer sequence. The Reverse Compiler figure illustrates this process; note that a slice of typing-related Ringer statements are mapped to a single Helena statement: `type "H" in p1`.

Each Helena statement stores its Ringer sequence, which the Helena interpreter runs to execute the statement. Thus the Reverse Compiler's output is a loop-free Helena program that runs the same underlying Ringer program it received as input.

Relation Selector



Problem Statement: To support a single-demonstration interaction model, we must determine for each recorded action (i) whether the action should be repeated on additional webpage elements and (ii) if yes, which additional webpage elements. In short, we must find relations relevant to the task.

Our Solution: A relation selection algorithm to extract information about relations from a straight-line interaction script.

¹This is a slight simplification. Statements for DOM events that are invisible to human users (e.g., `focus`), can have different n .

```

1 curSize = |N|
2 while curSize > 0 do
3   S = subsetsOfSize(N, curSize)
4   for subset in S do
5     rel = relationExtractor(w, subset)
6     if rel then
7       return rel
8   end
9   curSize = curSize - 1
10 end

```

Algorithm 1. Relation Selector algorithm.
Input: w , a webpage, and N , a set of interacted nodes in w . Output: rel , a relation of nodes in w that maximizes:
 $|\{n : n \in N \wedge n \in rel[0]\}|$

The central goal of our Relation Selector is to predict, based on a single loop-free replay script, whether the script interacts with any relations over which the user may want to iterate. Our Relation Selector approach has two key components:

- **Relation Selector:** The top-level algorithm. It takes a replay script as input and produces a set of relations as output.
- **Relation Extractor:** A subroutine. It takes a set of nodes as input. If it can find a relation that has one column for each input node, it returns the relation; if not, it returns null.

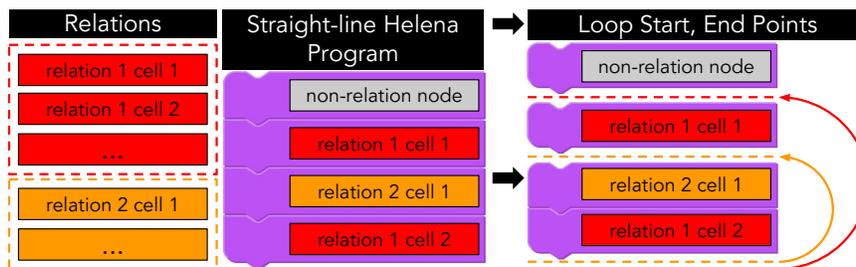
Relation Selector. The Relation Selector uses a feedback loop with the relation extractor. It starts by identifying the set of all DOM nodes with which the input replay script interacts, then groups them according to the webpages on which they appear. For each webpage w and its set of nodes N , the Relation Selector applies Algorithm 1. Essentially the Relation Selector repeatedly calls the relation extractor on subsets of N , from the largest subsets to the smallest, until it finds a subset of nodes for which the relation extractor can produce a relation. For instance, if a user interacted with a movie title, movie rating, and webpage title on a single webpage, the Relation Selector would first seek a relation that includes all three of those nodes in the first row. If the relation extractor fails to find a relation (the likeliest outcome given standard web design patterns), the Relation Selector tries subsets—e.g., subsets of size two until it finds a well-structured relation that includes the movie title and movie rating (but not the page title) in the first row.

Relation Extractor. Our custom relation extractor is closely related to the relation extractors [90, 50, 37, 32] discussed in Related Work, with one key difference: it is designed to excel in the case of having only one row of data. We found that prior relation extractor techniques often required at least two rows of data as input. Since being able to suggest a good relation given only a single row of data is critical to our interaction model, we developed a custom relation extractor adapted to this constraint. The key insight is to fingerprint the structure of the input cells' deepest common ancestor (DCA), then find a sibling of the DCA that shares the structure fingerprint (see illustration in Appendix 3.7). For instance, if the path through the DOM tree from the DCA n to an actor name is $p1$ and the path from n to the actor's role is $p2$, the relation extractor would seek a sibling of n that also has descendant nodes at the $p1$ and $p2$ positions. Using the sibling node as a second row of labeled cells, we can apply the same techniques that drive prior relation extractors.

Saved Relations. A centralized server stores a database of relation extractors used in past Helena programs. When the Relation Selector chooses which relation to extract from a page, it

considers both freshly identified relations and stored relations that originated on a related page and function on the current page. It ranks them, preferring relations that: include as many of the input nodes as possible, have many rows on the current page, and have been used in prior scripts. With this approach, users can expect that Helena already knows how to find relations on many mainstream sites.

Generalizer



Problem Statement: Scraping hierarchical data requires nested loops, with one loop for each level of the hierarchy.

Our Solution: A Generalizer that introduces nested loops.

The Generalizer takes a loop-free Helena program and a set of relations as input. It has three roles: (i) adapt the program to store output data, (ii) for each relation, determine where to insert a loop over the relation, and (iii) parameterize the loop bodies to work on loop variables rather than concrete values.

Output Statement. The input program collects data from DOM nodes but does not store the data or accumulate any output. The Generalizer appends an output statement at the tail of the Helena program to add a row of data to an output dataset; it adds a cell in the output statement for each scrape statement in the program. Thus a program that scrapes a movie title and an actor name produces [movie, actor] rows. If the actor is scraped in a loop, the program produces multiple rows with the same movie but different actors. This builds a ‘universal relation’ view of the hierarchical data, a join over all relations. This meets our data science collaborators’ requests for tabular data that they can manipulate with familiar spreadsheet and statistical software.

Loop Insertion. The input program is loop-free, but the output program should iterate over all relations identified by the Relation Selector. To add loops, the Generalizer first identifies whether each Helena statement’s target node appears in any relations. To identify the hierarchical structure, we produce a total order over relations based on the order in which their nodes appear in the program. The first relation whose cells are used is highest in the hierarchy (associated with the outermost loop); the relation that sees its first cell use only after all other relations have been used is lowest in the hierarchy (associated with the innermost loop). For each relation in the hierarchy, we insert a loop that starts immediately before the first use of the relation’s cells and ends after the output statement.

Parameterization. The input program operates on cells of the first dataset row, so after loop insertion, each loop body still only operates on the first row. For example, if the goal is to scrape 100 movies, the program at this intermediate point scrapes the *first* movie 100 times. The Generalizer adapts loop bodies so that we can apply them to multiple rows of a relation. We use *parameterization-by-value*, a metaprogramming technique for turning a term that operates on a concrete value into a function that can be called on other values. Essentially, $(pbv\ term\ value) \rightarrow (\lambda(x)\ term')$, where $term'$ is $term$ with all uses of $value$ replaced with uses of a fresh variable x . We execute parameterization-by-value for DOM nodes, typed strings, and URLs. For each Helena statement in a newly inserted loop, we check whether the target node appears in the loop's associated relation. If yes, we identify the index i of the target node n in the relation row. We then replace the Helena statement's slice of Ringer events E with $(pbv\ E\ n)(row[i])$, where row is the variable name used to refer to the relation row at each iteration. We repeat this process for typed strings, for each type statement in a loop (checking whether the typed string includes the text of any relation node), and for URLs, for each load statement in the loop.

Range

Data Shape. The depth of a Helena program's deepest nested loop is the bound on the depth of the collected hierarchy; our Generalizer can add any number of nested loops, so Helena can collect arbitrarily deep hierarchies. Sibling subtrees can have different depths—e.g., a movie can have no actors and thus no child nodes. The number of child nodes of any given node is unbounded and can vary—e.g., one movie can have 10 actors while another has 20. Thus, the number of dataset rows is unbounded. The number of columns per row is bounded by the number of scrape statements in the program; recall the Generalizer adds an output statement that adds rows with one cell for each scrape statement—e.g., if the user scraped a movie title and an actor name, each output row includes a movie title and an actor name. To collect variable-length data, users should add additional layers in the data hierarchy rather than variable-length rows—e.g., if users want a list of actors for each movie, they collect many [movie, actor] rows rather than one variable-length [movie, actor1, actor2,...] row per movie. The number of scrape statements bounds but does not precisely determine the number of populated cells per row; Helena leaves an empty cell for missing data—e.g., if an actor appears in a cast list without an associated character name, the role cell is left empty.

Node Addressing. “Node addressing” or “data description” is the problem of identifying the node on which to dispatch an action or from which to extract data. This is a long-standing problem in the web PBD community and largely determines how well a program handles page redesigns and other changes. Helena uses Ringer for replay and thus uses Ringer's node addressing (see [9]). This means any webpage changes that Ringer cannot handle, Helena also cannot handle.

Ambiguity. Helena uses a single demonstration as input to the PBD process, so inputs can be ambiguous. For instance, say a user scrapes a table in which some rows are user-generated posts and some rows are ads. The user may want to scrape all rows or only rows that have the same type as the demonstrated row. A single-row demonstration is insufficient to distinguish between these two cases. Thus it is critical that users have the option to edit output programs; this motivated making the Helena language so high-level and using a blocks-based editor (see output in Fig. 3.3). Although this

chapter focuses on the learnability of Helena’s PBD interaction, studying the editability of its output programs is a natural next step and critical to understanding whether Helena’s ambiguity-embracing approach is practical for real users.

3.6 Conclusion and Future Work

Helena’s novel interaction model and generalization algorithms push PBD web scraping beyond the restrictions that prevented past tools from collecting realistic datasets—in particular, distributed, hierarchical data. With a learning process at least 8x faster than the learning process for traditional scraping, this strategy has the potential to put web automation tools in the hands of a wider and more diverse audience. We feel this motivates two key future directions: (i) Although Helena is designed to combine PBD with a learnable, editable programming language, this chapter focuses on the program drafting interaction alone. Evaluating the editability of Helena programs is also critical. (ii) Helena was designed with social scientists—and end users generally—in mind. Because this work compares PBD to traditional programming, our evaluation focused on a population that can use traditional languages. We see testing with end users as a crucial next step on the path to democratizing web data access.

3.7 Relation Selector Example

Here we expand on the sample Relation Selector execution described in the Algorithms section. Say the user interacts with a webpage with the partial DOM tree depicted in Fig. 3.6 and scrapes the nodes highlighted in red: `page title`, `movie 1 title`, and `movie 1 rating`.

The Relation Selector starts by passing all three interacted nodes to the relation extractor, which tries to find a relation with all three nodes in the first row. The deepest common ancestor of all three is `page body`. The relation extractor identifies a structure fingerprint for `page body`; the fingerprint is a representation of the Fig. 3.6 edges highlighted in orange and purple—that is, the paths to all three interacted nodes. Next it looks for a sibling of `page body` that has nodes at all of those positions. Since `page header` does not have nodes at those positions, the relation extractor fails to find an appropriate sibling node (representing a predicted second row) for this set of input nodes. The relation extractor returns null for this input.

Next the Relation Selector tries subsets of the three interacted nodes, starting with subsets of size two. Eventually it tries `{ movie 1 title, movie 1 rating }`. For this subset, the deepest common ancestor is `movie 1`; its structure fingerprint includes the paths highlighted in purple. The relation extractor seeks a sibling of `movie 1` that has nodes at those paths and finds that `movie 2` has nodes `movie 2 title` and `movie 2 rating` at those positions. Since this subset produces a first row that has an appropriate sibling for making a second row, the relation extractor passes the first and second rows to a traditional relation extraction algorithm ([90, 50, 37, 32]), and the Relation Selector returns the result.

For simplicity, we do not describe cases in which Helena extracts multiple relations from one page, but this is necessary in practice.

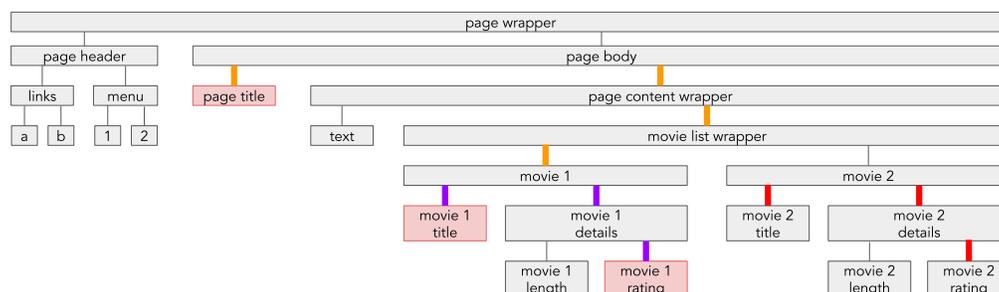


Figure 3.6: A sample DOM tree. If the user interacts with the nodes highlighted in red, the Relation Selector will produce a relation with one column for movie titles and one column for movie ratings. The relation extractor will find that `movie 1`'s structure fingerprint (the paths highlighted in purple) matches the structure fingerprint of `movie 2` (the paths highlighted in red).

Chapter 4

Programming Model Usability Evaluation

To evaluate the Helena PBD tool, we conducted a within-subject user study comparing Helena with Selenium, a traditional web automation library. Based on Design Goal D3, we were more interested in the learnability than the usability of the tool, so we focused our study on the following research questions:

- RQ1: Can first-time users successfully learn and use Helena to scrape distributed hierarchical data?
- RQ2: Will it be easier to learn to complete this task with Helena or with a traditional web automation language?

We recruited 15 computer science graduate students (9 male, 6 female, ages 20 to 38) for a two-hour user study. All participants had been programmers for at least 4 years and had used Python (one of Selenium’s host languages).

Procedure

We started each session with a description of the participant’s assigned web scraping task. Each participant was assigned one of two tasks: (i) Authors-Papers: Starting at Google Scholar, iterate through a list of authors, and for each author a list of papers. The authors list is a multi-page list; each author links to a profile page that lists papers; more papers can be loaded via a ‘More’ button that triggers AJAX requests. (ii) Foundations-Tweets: Starting at a blog post that lists charitable foundations, iterate through charitable foundations, and for each foundation a list of tweets. The foundations list is a single-page list; each foundation links to a Twitter profile that lists tweets; more tweets can be loaded by scrolling the page to trigger AJAX requests. Authors-Papers came from [16], and Foundations-Tweets from a Public Policy collaborator.

Next, we asked each participant to complete the assigned task with two tools: Helena and Selenium [83] (in particular, the Python Selenium library). The tool order was randomized to distribute the impact of transferable knowledge. For each tool, we pointed the user to a webpage with documentation of the tool but left the choice about whether and how to use that documentation

(or any other resources available on the web) up to the participant; being primarily interested in learnability, we wanted to observe the time between starting to learn a tool independently and understanding it well enough to complete a task with it, so we let participants control the learning experience. If a participant did not complete the task with a given tool within an hour, we moved on to the next stage.

After using both tools, each participant answered a survey that asked them to reflect on the tools' learnability and usability.

Comparison Against Traditional Programming. By conducting a within-subject user study, we obtain a fair evaluation of the productivity benefits of PBD scraping versus traditional scraping for realistic datasets. However, this design choice limits our participants to coders—novices with Helena, but not novice programmers. Note that a comparison against state-of-the-art PBD web automation is not possible, since no existing PBD tool can scrape the distributed hierarchical datasets we use in our tasks.

Results

User Performance

Fig. 4.1 shows completion times. Recall that since we are most interested in learnability, which we operationalized as time between starting to learn a tool and producing a first correct program with it, we did not distinguish between time spent learning and time spent writing programs. Thus, completion times include both learning and program authoring time.

All participants completed their tasks with Helena, for a completion rate of 100%. In contrast, only four out of 15 participants completed their tasks with Selenium, for a completion rate of 26.7%. Helena was also fast. All participants learned and successfully used Helena in under 10 minutes, four in less than 5 minutes. The median time was 6.67 minutes.

For the Authors-Papers task, the average completion time with Helena was 5.7 minutes. With timed-out participants' times truncated at 60 minutes, the average completion time with Selenium was 58.0 minutes. For the Foundations-Tweets task, the average completion time with Helena was 7.3 minutes. With timed-out participants' times truncated at 60 minutes, the average completion time with Selenium was 54.7 minutes. Because our data is right-censored, with all Selenium times above 60 minutes known only to be above 60 minutes, we cannot provide a precise estimate of the learnability gains for Helena versus Selenium. However, we can offer a lower bound: the time to learn Selenium well enough to complete our tasks is at least 8.5x higher than the time to learn Helena.

Although excluding right-censored data leaves only four data points, Helena's effect on performance is statistically significant even for this small sample. We conducted a paired-samples t-test for participants who completed the task with both tools. There was a significant difference in the completion times for the Selenium ($M=2751.2$, $SD=600.2$) and Helena ($M=405.8$, $SD=173.1$) conditions; $t(3)=8.534$, $p = 0.0034$.

To evaluate timed out participants' progress towards a working Selenium scraper, we defined five checkpoints on the way to a complete scraper. If a user completed all checkpoints we, considered

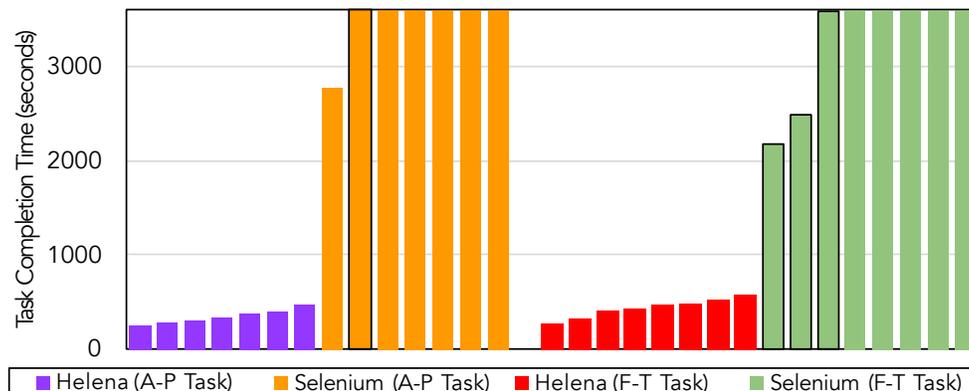


Figure 4.1: Combined learning and task completion times for the Authors-Papers and Foundations-Tweets tasks, using Helena and Selenium. The cutoff was one hour; Selenium bars that extend to the one hour mark indicate the participant was unable to complete the task with Selenium. All participants succeeded with Helena in under 10 minutes. Only four of 15 succeeded with Selenium. Four Selenium bars are marked with a black outline, indicating the participant had prior experience with Selenium. No participant had prior experience with Helena.

the participant to have completed the task, although there are in fact more steps required to make the programs robust and failure-tolerant (like the Helena programs). We found that six of 15 participants never reached any of the checkpoints with Selenium. See Table 4.1 for a record of which checkpoints each participant reached.

We also measured the number of debugging runs as an additional measure of work. On average, when participants used Helena, they ran 0.2 debugging runs. (Of 15 participants, 13 only ran a correct program and thus had no debugging runs.) On average, using Selenium, they ran 20.8 debugging runs. Again, Selenium values are right-censored, since participants who did not finish might need more debugging runs.

Although all participants were first-time Helena users, 4 of 15 participants had used Selenium in the past. Those with prior Selenium experience accounted for 3 of the 4 participants who completed their tasks with Selenium; one participant with prior Selenium experience was unable to complete the task, and only one participant with no prior Selenium experience was able to complete a task with Selenium.

User Perceptions

We were interested in whether our participants, being programmers, would prefer PBD or a traditional programming approach. In the survey after the programming tasks, we asked participants to rate how usable and learnable they found the tools. Participants answered all questions with a seven-point Likert scale, with 1 indicating the tool was very easy to use or learn and 7 indicating the tool was very difficult to use or learn. The average ratings for Helena and Selenium *usability* were 1.2 and 4.8, respectively. The average ratings for Helena and Selenium *learnability* were 1.1

Task	1 full row	1st pg outer loop	2nd pg outer loop	1st pg inner loop	2nd pg inner loop
A-P					
A-P					
A-P					
A-P	✓				
A-P	✓	✓		✓	
A-P	✓	✓		✓	
A-P	✓	✓	✓	✓	✓
F-T			-		
F-T			-		
F-T			-		
F-T		✓	-		
F-T		✓	-		
F-T	✓	✓	-	✓	✓
F-T	✓	✓	-	✓	✓
F-T	✓	✓	-	✓	✓

Table 4.1: Because we wanted to understand partial completions of tasks with Selenium, we defined subtasks of the Authors-Papers and Foundations-Tweets tasks. We imposed higher standards for Helena programs, but for the Selenium component of the study, we considered a task complete once the participant wrote a script that completed all five of these subtasks. The subtasks are: (i) producing a row with all requisite cells, (ii) iterating through the first page of the relation associated with the outer loop, (iii) iterating through at least the first two pages of the relation associated with the outer loop, (iv) iterating through the first page of the relation associated with the inner loop, and (v) iterating through at least the first two pages of the relation associated with the inner loop. Note that the Foundations-Tweets task has only four subtasks because the list of foundations appears entirely in a single webpage; we show ‘-’ in the ‘2nd pg outer loop’ column to indicate there is no second page of data for this loop. Overall, less than half of the participants wrote a script that could collect one complete row of the target dataset within the one-hour time limit.

and 5.6, respectively.

We also asked what tools participants would want for future scraping tasks of their own. Participants could list as many tools as they liked, including tools they knew from elsewhere. All but one participant (93.3%) indicated they would use Helena; two (12.5%) indicated they would use Selenium.

User Study Discussion

What is challenging about traditional web automation? To get a range of views on what makes traditional web automation difficult, we asked participants what was hardest about using Selenium. The answers ranged from “Everything” to “Selecting elements on a webpage designed by someone else; creating new windows or tabs (supposedly possible...); expanding out paginated data tables.” to “Mystifying browser behaviors (are we waiting for a page to load? is an element visible? is the element findable but not clickable????)” to “restricted low-level interactions (e.g., navigation and element selection).” to “I had trouble finding relevant portions of the API (methods

for WebElements for example). Also, I relied on Google/Stack Overflow for debugging.” to “I worry that my Selenium script is not very robust: I might have missed a lot of cases or used ambiguous selectors to find elements. It will also be much harder to update my Selenium script if Google Scholar ever changes their HTML structure.”

What is challenging about PBD web automation? In contrast, the most common answer to what was hardest about using Helena was some variation of “Nothing” or “NA”; nine out of 15 participants provided these answers. As we expected, the primary concerns were about control: “to what extent would people be able to tweak things outside of the automatic framework?”, “I think if I got used to using Selenium regularly, I would feel limited by Helena.” This impression may be the effect of our user study design focusing on Helena’s drafting interface rather than the editing interface—the assigned tasks did not, for instance, require users to add if statements, even though Helena supports this. On the other hand, participants could not think of web scraping tasks they would want to accomplish that could not be handled with Helena: “Selenium is good for more complex tasks, I guess, but I cannot think of anything what Selenium can do while Helena cannot, given you can change the script manually after Helena creates the script.”

What is good about traditional web automation? As lack of low-level control was considered a weakness of PBD web automation, it was considered a strength of traditional strategies: “More manual work so maybe more flexibility (?)”, “Very fine grained control if you have the time to figure it all out.” Others suggested Selenium might be more useful in the context of broader programming tasks: “The resulting script could be added as part of a larger software pipeline” or “you could run it headless and in parallel on multiple computers.” This suggests it is easier for programmers to imagine programmatically manipulating a Selenium program than a Helena program, although the suggested manipulations – using a scraping program within a larger pipeline or running it in a distributed setting – are indeed possible (and already tested) with Helena. Finally, one respondent suggested using Selenium is good because it is a fun puzzle: “I love programming...it is like a puzzle for me. So the fact that Selenium is an API and I need to write code to use it, is fun for me personally :)”

What is good about PBD web automation? Participants appreciated the fact that Helena handled the hierarchical structure for them (“[It] was very useful how it automatically inferred the nesting that I wanted when going to multiple pages so that I didn’t have to write multiple loops.”) and how it shielded them from low-level node finding and relation finding (“Locating the thing I care about is much simpler: just click on it”). They also felt it anticipated their needs: “I didn’t know anything about web scraping before starting and it fulfills a lot of the functionality I didn’t even expect I would need,” “Super easy to use and I trust that it automatically follows the links down. It felt like magic and for quick data collection tasks online I’d love to use it in the future.”

How do traditional and PBD web automation compare? We also solicited open-ended responses comparing Helena and Selenium, which produced a range of responses: “The task was an

order of magnitude faster in Helena,” “Helena is much, much quicker to get started with,” “If I had to use Selenium for web scraping, I would just not do it,” “Helena’s way easier to use – point and click at what I wanted and it ‘just worked’ like magic. Selenium is more fully featured, but...pretty clumsy (inserting random sleeps into the script),” “Helena is a self balancing unicycle, Selenium is a 6ft tall unicycle. One you can buy/download and [you’re] basically up and going. The other you needs years of practice just to get to the place where you feel comfortable going 10 ft.” We were interested in what comparisons participants chose to make. Ultimately, the comparisons sorted from highest frequency to lowest frequency were about: programming time, ease-of-use, ease-of-learning, language power, and program robustness.

Chapter 5

Skip Blocks for Failure Recovery, Incrementalization, and Handling Data Churn

With more and more web scripting languages on offer, programmers have access to increasing language support for web scraping tasks. However, in our experiences collaborating with data scientists, we learned that two issues still plague long-running scraping scripts: i) When a network or website goes down mid-scrape, recovery sometimes requires restarting from the beginning, which users find frustratingly slow. ii) Websites do not offer atomic snapshots of their databases; they update their content so frequently that output data is cluttered with slight variations of the same information — *e.g.*, a tweet from profile 1 that is retweeted on profile 2 and scraped from both profiles, once with 52 responses then later with 53 responses.

We introduce the *skip block*, a language construct that addresses both of these disparate problems. Programmers write lightweight annotations to indicate when the current object can be considered equivalent to a previously scraped object and direct the program to skip over the scraping actions in the block. The construct is hierarchical, so programs can skip over long or short script segments, allowing adaptive reuse of prior work. After network and server failures, skip blocks accelerate failure recovery by 7.9x on average. Even scripts that do not encounter failures benefit; because sites display redundant objects, skipping over them accelerates scraping by up to 2.1x. For longitudinal scraping tasks that aim to fetch only new objects, the second run exhibits an average speedup of 5.2x. Our small user study reveals that programmers can quickly produce skip block annotations.

5.1 Introduction

About a year before the work described in this chapter, we started working with a team of sociologists investigating how rents were changing across Seattle neighborhoods. They aimed to scrape Craigslist apartment listings once a day. We met with the research assistant who would handle data collection and walked him through using the Helena scraping tool. We expected that the main challenges

would be scraping script errors, errors like failing to extract data from some webpages. Instead, the biggest obstacle was that the machine he used for scraping—a laptop connected to his home WiFi network—regularly lost its network connection in the middle of the night, partway through long-running scrapes. He would check progress the following mornings, find that the network connection had gone down, then have to restart the script. Restarted executions repeated much of the originals’ work, and they were no less vulnerable to network failures.

Worse, the scraping script was repeating work even when the network was well-behaved. Craigslist shows a list of ads that relate to a search query. Each page of results shows a slice of the full list. To select the slice for a new page, Craigslist indexes into the up-to-date search result. The result is updated whenever a new ad is posted, with new ads displayed first — that is, appended to the head of the list. If a new ad is posted between loading page n and loading page $n + 1$, the last ad from page n shifts to page $n + 1$. In a scraped snapshot of the list, the ad appears twice. With new rental listings arriving at a fast pace, a single execution of the scraping script routinely spent 13 hours rescraping previously seen ads.

These are not errors in the scraping script *per se* — if the network were perfect and the website served an atomic snapshot of its data, the scraping script would have collected the data correctly. The importance of network and server failures was an unexpected finding that changed our perspective on how to improve the scraper-authoring experience. This chapter presents the result, the design of language support to address extrinsic challenges that hamper long-running web scrapers when they collect large, real datasets.

Extrinsic Challenges to Long-Running Scraping Tasks

Our experience suggests a programming model for large-scale web scraping must handle these four challenges:

1. *Web server and network failures.* Server outages and temporary network connection failures can interrupt a scraping script and terminate the script’s server session. In general, it is impossible to resume execution at the failure point. While we could checkpoint the client-side script and restart it at the last checkpoint, we do not control the web server and thus cannot restart the server session at the corresponding point. Section 5.4 discusses recovery strategies, concluding that restarting execution from the beginning appears to be the only general option. However, scraping scripts that run for hours or days are likely to fail again if the mean time to failure of a data scientist’s WiFi connection is only a few hours.
2. *Alternative data access paths.* It is common that some website data can be accessed via different navigation paths. For example, consider scraping the profiles of all Twitter users retweeted by one’s friends. If two friends have retweeted user A, we will scrape A’s user profile at least twice. Rescraping can produce inconsistent data: one data point indicating that A has 373 followers and one indicating he has 375 followers. Repeated and inconsistent entries could be merged in postprocessing, but duplicates interfere with analytics that we may want to perform as the data is collected. Further, revisiting profiles can substantially slow the execution.

3. *Non-atomic list snapshots.* A script typically cannot obtain atomic snapshots of website databases because they can be updated during script execution, which changes the web server’s output to the browser. This includes cases like the Craigslist ad updates that cause some ads to appear on multiple pages of listings and ultimately to appear multiple times in the scraped snapshot. As with duplicates caused by multiple access paths, this may produce internally inconsistent data. Additionally, rescraping this kind of duplicate wastes up to half of scraping time (see Figure 5.6a).
4. *Repeated scraping.* Even when a script finishes without failure, there can be reasons to rerun it. First, the non-atomic snapshot phenomenon may cause data to be skipped during pagination. (For instance, in the Craigslist example, deleting an ad could move other ads up the list.) Rerunning the script increases the likelihood that all data is collected. Second, we may want to scrape a new version of the dataset days or weeks later to study trends over time. In both scenarios, it is common for most data on the website to remain unchanged, so much of the rescraping is redundant. Empirically, we observe that in our users’ datasets about 74% of scraped data was already present one week before.

The Skip Block

We address all four challenges with a single construct called the *skip block*. The skip block wraps script statements that scrape a logical unit of data — for example, the profile of a single user. When the skip block finishes without a failure, it remembers that the profile has been scraped. When this profile is encountered again, the block is skipped. The rest of this subsection explains some key features of skip blocks.

User-defined memoization. How does the skip block know it is safe to skip the block’s statements? After all, the website data may have changed since it was last scraped, in which case the statements would produce new values. In general, it is impossible to ascertain that the block would produce the same values without actually re-executing the entire block. We must therefore relax the requirement that the scraped values are identical. The idea is to involve the user, who defines the “key attributes” for each skip block. For instance, the key could be the triple (*first name, last name, current city*) for a user profile. When this skip block scrapes a triple that has previously been scraped, it skips the statements that would scrape the remaining attributes of the user, even though they may have changed.

Hierarchical skip blocks. Skip blocks can be nested, which gives us adaptive granularity of skipping. For example, cities have restaurants, and restaurants have reviews. If we want to scrape reviews, we may want to put skip blocks around all three entity types: reviews, restaurants, and cities. These skip blocks will be nested. For this task, as with many large scraping tasks, a single skip block is insufficient. Imagine we use skip blocks only for the restaurant entity, and the network fails during script execution. If the failure occurs after all of Berkeley’s restaurants were scraped, the script will still have to iterate through the list of thousands of Berkeley restaurants before it can proceed to the next city. In contrast, if we use a skip block at the city level, the script will simply skip over all of Berkeley’s restaurants and their associated reviews, proceeding directly to the next

city. However, a single skip block at the city level is also impractical. Scraping a single city's reviews takes hours. If the failure occurs while visiting the 2,000th Berkeley restaurant, we do not want to throw away the hours of work that went into scraping the first 1,999 Berkeley restaurants. By using a skip block at both the city and the restaurant level, we can handle both of these failures gracefully, skipping whole cities where possible but also falling back to skipping over restaurants.

Staleness-controlled skipping. Programmers can override skipping of previously scraped data. This control relies on timestamps stored when the data is scraped. We use two kinds of timestamps: (1) *Physical timestamps.* Say we are scraping the restaurant example once a week and want to continually extend the scraped dataset with fresh website updates. In this scenario, the programmer may specify that a restaurant should be skipped only if it has been (re)scraped in the past month. This means that the data will be refreshed and new reviews retrieved if the most recent scrape of the restaurant is more than a month old. (2) *Logical timestamps.* To recover from a failure — e.g., a network failure — we skip a block if it was collected since the prior *full run*, which we define as an execution that terminated without failure. To handle this scenario, we use logical timestamps. Logical time is incremented after each full run.

Implementation. We have added skip blocks to the Helena web scripting language. Skip blocks could also be introduced to other web scripting languages, such as Selenium [83], Scrapy [80], and BeautifulSoup [73]; see Section 5.5 for a discussion of alternative implementations of skip blocks.

We selected Helena because it was designed for end users who author Helena programs by demonstration. We believe that access to skip blocks may benefit end users even more than programmers because the latter can overcome the problems that motivate skip blocks by reverse-engineering the website and devising website-specific solutions.

In contrast, the user of a Programming by Demonstration (PBD) scraping tool may lack (i) the programming skills or (ii) the insight into webpage structure necessary to build a site-specific fix. Without explicit language support, many users will have no recourse when network and server failures arise. Skip blocks offer the advantage of being easy for end users to write (see Section 7) and also being applicable regardless of the underlying webpage structure, so that end users can apply them wherever our motivating failures emerge.

Evaluation

We evaluate skip blocks using a benchmark suite of seven long-running scraping scripts that scrape datasets requested by social scientists. Our evaluation assesses how skip blocks affect failure recovery (Challenge (1)), a single execution (Challenges (2) and (3)), and multiple executions (Challenge (4)). We also evaluate whether skip blocks are usable.

We evaluate the effects of skip blocks on failure recovery and find that they prevent scripts from unnecessarily rescraping large amounts of data, accelerating recovery time by 7.9x on average. The average overhead compared to a hypothetical zero-time recovery strategy is 6.6%. We also evaluate the effects of skip block use on longitudinal scraping tasks. We repeat a set of large-scale scraping tasks one week after a first run and observe that using the skip block construct produces speedups between 1.8x and 799x, with a median of 8.8x.

Finally, we evaluate whether programmers and non-programmers can use the skip block construct with a small user study. We find that users can both learn the task and produce eight out of eight correct skip block annotations in only about 12 minutes.

Contributions

We make these novel contributions:

- *The skip block, a construct for hierarchical memoization with open transaction semantics.* The programming model (i) allows users to define object equality, specifying which object attributes to treat as volatile and non-identifying and (ii) adapts the memoization granularity during recovery, skipping initially large blocks, then many smaller blocks.
- *A notion of staleness that handles both failure recovery and longitudinal scraping.* With timestamp-based staleness constraints, a script skips a block only if it was previously committed within a specified interval of physical or logical time.
- *A suite of web scraping benchmarks and an evaluation of skip blocks.* We developed a benchmark suite of large scraping tasks. We evaluated how skip blocks affect performance on these benchmarks in three scenarios: failure recovery, a single-execution context, and a multi-execution context. We also conducted a user study that assesses the usability of our construct.

In this chapter, Section 5.2 expands on the rationale for our skip block design. Next, we give the semantics of the skip block construct in Section 5.3. Section 5.4 discusses the feasibility of alternative designs. Section 5.5 details the implementation. Our evaluation includes details of our benchmark suite (Section 5.6), then evaluations of both the performance implications of skip blocks (Section 5.7) and whether they are usable by end users (Chapter 7). Finally, Section 5.8 discusses how our approach relates to prior work.

5.2 Overview

In this section, we will first categorize the four scraping challenges that motivate our new language construct into two failure classes. Next, we will discuss the key characteristics of the skip block construct, then how these characteristics allow skip blocks to handle the two failure classes.

Web Scraping Failures

During the course of developing Helena and testing its expressivity, we shared the PBD tool with several groups of social scientists and a handful of computer scientists. Following these experiences and the feedback we received during script development, we put together a list of the failures and programmability issues that we had observed. While this review of failure types was unsystematic,

based mostly on users' requests for help, it revealed an interesting pattern. The most serious failures, the four described in Section 5.1, fell into two categories:

- **Access failures:** We use *access failure* to refer to any failure that prevents a server request from producing the expected response in the web browser. This includes cases where the scraper loses the network connection and also cases where the server goes down or otherwise ceases to respond. This also extends to cases in which a session times out or otherwise ends, requiring the user to log back into a site.
- **Data consistency failures:** We use *data consistency failure* to refer to any failure caused by the fact that web server output does not come from a single atomic read of the website's internal state. Websites may update their underlying data stores between page loads within a single execution of a scraping script and also between executions. They may add new objects to lists of objects, reorder objects in a list, or change object attributes. The end result is that objects can fail to appear in a given script execution or they can appear multiple times, sometimes with different attribute values.

Another look at the challenges described in Section 5.1 reveals that Challenge (1) stems from access failures while Challenges (2), (3), and (4) are different manifestations of data consistency failures.

Access and data consistency are classes of important failures that no client-side scraping program can prevent, since they are extrinsic to the scraping process. Even the best custom scraper cannot prevent the server from going down or the site from updating its database between two page loads. Given that these failures cannot be eliminated, it became clear we needed to offer Helena programmers a way to handle them gracefully.

Running Example

To explore the key characteristics of the skip block construct, we will use a running example. Say we want to scrape information about all papers by the top 10,000 authors in Computer Science, according to Google Scholar. We will iterate through each author. For each author, we will iterate through the author's papers. Finally, for each paper, we will iterate through the histogram of the paper's citations by year.

Figure 5.1 shows a program that executes this task using the Helena web automation language. Helena offers programmers abstractions for webpages, logical relations (e.g., authors, papers, years), cells of the relations, and DOM nodes. Relations are logical in that their rows may be spread across many pages, which users typically navigate using 'Next' or 'More' buttons. In our example, the authors relation is a list of 10,000 (author_name) tuples spread over 1,000 webpages, because each page of results shows only 10 authors; there are many papers relations, one per author, and a typical papers relation is a list of hundreds of (title, year, citations) tuples, spread over 10-50 pages.

In Figure 5.1, variables p1, ... , p4 refer to the pages on which actions are executed. The expression p2 . authors extracts rows of the authors relation from page p2. (Note that Helena hides

```

1  p1 = load("https://scholar.google.com/citations?view_op=search_authors")
2  type(p1, , "label:computer_science") // type query into Google Scholar
   search bar
3  p2 = click(p1, ) // click on Google Scholar search button
4  for author_name in p2.authors{
5      p3 = click(p2, author_name)
6      h_index = scrape(p3, )
7      for title, year, citations in p3.papers{
8          p4 = click(p3, title)
9          for citation_year, citation_count in p4.citations{
10             addOutputRow([author_name, h_index, title, citations, year, citation_year,
11                          citation_count])
12         }
13     }

```

Figure 5.1: The Helena program for scraping papers by authors who are labeled with the ‘computer science’ tag in Google Scholar. Green-outlined boxes are screenshots of DOM nodes, which the Helena tool takes during a user demonstration phase and displays as user-facing identifiers for DOM nodes. The variables p1 through p4 are page variables, each referring to an individual webpage. The expression p2.authors in line 4 evaluates to a reference to the authors relation represented in page variable p2. Essentially, authors is a function that maps a page to the slice of author data in the page; it uses DOM paths and other node attributes to identify relation cells.

```

1  p1 = load("https://scholar.google.com/citations?view_op=search_authors")
2  type(p1, , "label:computer_science")
3  p2 = click(p1, )
4  for author_name in p2.authors{
5      p3 = click(p2, author_name)
6      h_index = scrape(p3, )
7      skipBlock(Author(author_name.text, author_name.link, h_index.text)){
8          for title, year, citations in p3.papers{
9              skipBlock(Paper(title.text, year.text), Author){
10                 p4 = click(p3, title)
11                 for citation_year, citation_count in p4.citations{
12                     addOutputRow([author_name, h_index, title, citations, year, citation_year,
13                                  citation_count])
14                 }
15             }
16         }
17     }

```

Figure 5.2: The Figure 5.1 program with skip blocks added. The first skip block (line 7) indicates that an author object should be skipped if a previously scraped author had the same name, link, and h-index. The second skip block (line 9) indicates that a paper object should be skipped if a prior paper with the same author ancestor also had the same title and publication year.

Table 5.1: A snippet of the dataset output by the Figure 5.1 and Figure 5.2 programs. The first two columns are associated with **authors**, the next three columns with **papers**, and the final two columns with **citation years**.

author_name	h_index	title	citations	year	citation_year	citation_count
J. Doe	34	A Paper Title	34	2016	2016	11
J. Doe	34	A Paper Title	34	2016	2017	23
J. Doe	34	Another Paper Title	28	2012	2012	9
...

the fact that rows of the relation are in fact spread across many pages.) DOM nodes are represented by names if they have been assigned names (as when they are associated with a cell in a relation) or by screenshots of the nodes. The `addOutputRow` statement adds a row to the output dataset. By default, the Helena PBD tool creates scripts that produce a single relation as output because the target users are familiar with spreadsheets. Our sample program produces an output dataset like the snippet of data in Table 5.1.

The Skip Block Construct

We briefly describe the key characteristics of the skip block construct.

Object Equality. Figure 5.2 shows a version of our running example that includes two skip blocks. The first skip block introduces an Author entity. It indicates that if two different author objects have the same name text, the same name link, and the same h-index text, we should conclude that they represent the same author. It also indicates that all code in the associated block scrapes data related to the author. The attributes for defining object equality can come from multiple different webpages, as they do in this case, with the author name appearing on p2 but the h-index on p3.

Committing. When the body of a skip block successfully finishes, we add a new record to a durable commit log. To register that an object’s skip block completed, the program commits the tuple of object attributes used to uniquely identify it — in our example, the name text, name link, and h-index text of the author.

Skipping. When we enter a skip block, we first check the commit log to see if we have already successfully scraped the current object. Recall that we conclude two objects represent the same entity if they share the key attributes identified by the programmer. If we find a commit record for the current object, we skip the body of the skip block. For instance, if we scrape author A in one execution, then come back and scrape the data the next day, we will probably see author A again. This time, as long as the name text and link are the same and the h-index is the same, the script will skip the associated block; even if the author’s email or institution have changed, the script will skip the block. However, say the author’s h-index is updated; in this case, the script will not skip the block, and it will rescrrape author A’s papers.

Nested skip blocks. The second skip block in Figure 5.2 introduces a Paper entity. This is a nested skip block, because it appears in the body of the author block. A nested block is skipped when its enclosing block is skipped. Thus, when an author block is skipped, all the associated paper blocks are also skipped. The paper block in Figure 5.2 indicates that if two paper objects have the same title

text, year text, *and* the same Author ancestor, we should conclude that they represent the same paper. The Author ancestor comes from the enclosing skip block. We could also define Paper without the Author argument, which would change the output dataset. Recall that paper data is DAG-structured. If author A and author B coauthor paper 1, it will appear in the paper lists of both A and B. If we encounter paper 1 first in A's list, then in B's list, we will have to decide whether to scrape it again in B's list. Should we skip it, because we only want to collect a list of all unique papers by the top 10,000 authors? Should we scrape it again because we want to know that both author A and author B have a relationship with paper 1? The Figure 5.2 definition is the right definition if we want to collect the set of relationships between authors and papers. The alternative definition — the one that does not require an Author match — is the right definition if we just want the set of all papers by the top 10,000 authors.

Atomicity. Note that the paper skip block includes an `addOutputRow` statement. This statement adds a row to the output — that is, the dataset the user is collecting — not to the commit log. However, because it is inside a skip block, this statement now behaves differently. The output rows are no longer added directly to the permanent data store. Rather, they are only added if the entire skip block finishes without encountering a failure. (Note that a failure terminates the whole program.) An `addOutputRow` statement belongs to its immediate ancestor. Thus, the Figure 5.2 `addOutputRow` adds data when the Paper skip block commits, not when the Author skip block commits.

Open Transactions. Inner skip blocks can commit even if ancestor skip blocks fail. In this sense they are akin to open nested transactions [62]. This design means that even if the network fails before the script commits author A, if it has scraped 500 of author A's papers, it will not revisit those 500 papers or their citations.

Timestamps. Each commit record is tagged with two timestamps, one physical and one logical. The logical timestamp is incremented each time the end of the scraping script is reached. This means that even if the script must restart to handle an access failure, all commit records produced before a successful script termination will share the same logical timestamp.

Staleness Constraints. The skip blocks in Figure 5.2 will skip if any run of the program has ever encountered the current object. If we want to change this behavior, we can provide a staleness constraint. By default, we use a staleness constraint of $-\infty$, which allows matches from anywhere in the commit log. However, we can provide an explicit staleness constraint in terms of either physical or logical timestamps. For instance, we could say to skip if we have scraped the current author object in the last year (`skipBlock(Author(author_name.text, author_name.link, h_index.text), now - 365*24*60)`), or we could say to skip if we have scraped the author object in the current scrape or any of our last three scrapes (`skipBlock(Author(author_name.text, author_name.link, h_index.text), currentExecution - 3)`).

Handling Web Scraping Failures with Skip Blocks

At a high level, skip blocks allow us to handle access failures by restarting the script and skipping over redundant work. In an ideal scenario, the server output after restarting would be the same, and it would be clear which work was redundant and thus how to return to the failure point. Unfortunately,

data consistency failures mean the server output may change, making redundancy hard to detect. Thus, the skip block approach handles consistency failures by allowing programmers to define redundancy in a way that works for their target webpages and dataset goals.

To illustrate how skip blocks handle individual access and data consistency failures, we walk through how skip blocks address the four challenges described in Section [5.1](#).

1. *Server and network failures.* Upon encountering an access failure, scripts restart at the beginning and use memoization-based fast-forwarding to accelerate failure recovery. With an author skip block in place, the script may assume that when the end of a given author's block has been reached, all his or her papers and all the papers' citations have been collected. Thus, if we encounter this same author during recovery, the script skips over visiting the many pages that display the author's paper list and the one page per paper that displays the paper's citation list. Since page loads are a primary driver of execution time, this has a big impact on performance.
2. *Alternative data access paths.* Skip blocks allow us to cleanly handle redundant web interfaces, skipping duplicates in DAG-structured data. Say we want to collect all papers by the top 10,000 authors in computer science. With the Google Scholar interface, we can only find this set of papers via the list of authors, which means we may encounter each paper multiple times, once for each coauthor. For our purposes, this web interface produces redundant data. By adding a skip block for the paper entity, we can easily skip over papers that have already been collected from a coauthor's list of papers.
3. *Non-atomic list snapshots.* Data updates during the scraping process can cause some data to appear more than once. Recall the Craigslist listings that move from the first page of results to the second when new items are added. This crowds the output dataset with unwanted near-duplicates and also slows the scrape. Each time we re-encounter a given listing, we must repeat the page load for the listing page. For objects with many nested objects (e.g., cities with many restaurants or restaurants with many reviews), the cost of repeats can be even higher. The skip block construct handles this unintentional redundancy. When an old listing is encountered again on a later page, the commit log reveals that it has been scraped already, and the script skips the object.
4. *Repeated scraping.* Naturally, the reverse of the Craigslist problem can also occur; websites can fail to serve a given relation item. (Imagine the second page of Craigslist results if no new listings were added but a first-page listing was deleted.) Sites generally have good incentives to avoid letting users miss data, but still this particular atomicity failure motivates the need to run multiple scrapes. Rescraping is also desirable for users who are conducting longitudinal collection tasks, or for users who simply want to keep their data snapshot up to date. The rescraping task is complicated by the fact that each type of scrape should proceed differently. If we are conducting an immediate scrape to ensure we did not lose Google Scholar authors in a mid-scrape ranking update, we certainly do not want to descend to the level of citations for all the authors already scraped — rather, we want to scrape only any authors we missed

in the last execution. If we are rescraping once a week, we are probably most interested in authors who are new to the top 10,000 list, but maybe we also want to refresh data for an author if the last time we scraped him or her was over a year ago. Our skip block staleness constraints allow us to offer both of these options. For the immediate rescrape, we would provide the last scrape’s logical timestamp as a staleness constraint. To rescrape all new authors and any author who has not been refreshed in the last year, we would provide the current time minus one year.

5.3 Semantics

This section presents the semantics of the skip block construct that Helena uses for detecting duplicate entities. First we describe the semantics without the details of staleness constraints. Next we show how the skip block rules change when we include staleness constraints.

Each skip block annotation takes the form:

$$\text{skipBlock}(\text{annotationName}(\text{attributes}), \text{ancestorAnnotations})\{\text{statements}\}$$

A sample annotation might look like this:

```
skipBlock(Paper(title.text, year.text), Author){ ... }
```

Figure 5.3 contains the semantics of the skip block construct. a ranges over node attributes. x ranges over entity scope variables. s ranges over statements. V_c is the store of currently active skip block vectors (a map $x \rightarrow \bar{a}$ from variable names to attribute vectors). V_p is the permanent store of completed skip block vectors (a map $x \rightarrow A$ from variable names to sets of attribute vectors). D_c is the store of output data staged to be added by the current skip block. D_p is the permanent store of the output data.

The COMBINEVECTORS rule handles skip blocks that depend on ancestor skip blocks. For instance, our sample skip block indicates that we will consider a paper a duplicate if it shares the title and publication year of another paper, but only if the Author (a skip block variable introduced by a preceding skip block) is also a duplicate. It concatenates the vector of the new skip block (\bar{a}) with the vectors of all ancestor skip block arguments ($\bar{a}_1 \dots \bar{a}_n$) and saves the combined vector as the current vector for the skip block (x).

The ALREADYSEENVECTOR and NEWVECTOR rules handle the case where the skip block has already been converted into a skipBlockHelper (which does not depend on ancestor skip blocks). The ALREADYSEENVECTOR rule handles the case where the input vector is already present in $V_p[x]$, the store of previous x input vectors—that is, the case where the skip block has identified a duplicate. In this case, the body statements (\bar{s}) are skipped.

The NEWVECTOR rule handles the case where the input vector is not present in $V_p[x]$ and thus x represents a new entity. In this case, we execute the body statements, then a commit statement.

The COMMIT rule saves the current x vector (\bar{a}) into the long-term store of x vectors. It also saves all dataset rows produced during the current skip block ($D_c(x_l)$) into the permanent dataset store (D_p).

$$\begin{array}{c}
 \frac{V_c(x_1) = \bar{a}_1 \quad \dots \quad V_c(x_n) = \bar{a}_n}{\langle \text{skipBlock}(x(\bar{a}), x_1 \dots x_n) \{ \bar{s} \}, V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skipBlockHelper}(x(\bar{a} \ \bar{a}_1 \dots \bar{a}_n)) \{ \bar{s} \}, V_c[x := \bar{a}], V_p, D_c, D_p, \bar{x}_c \rangle} \text{ COMBINEVECTORS} \\
 \\
 \frac{\bar{a} \in V_p[x]}{\langle \text{skipBlockHelper}(x(\bar{a})) \{ \bar{s} \}, V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skip}, V_c[x := \text{null}], V_p, D_c, D_p, \bar{x}_c \rangle} \text{ ALREADYSEENVECTOR} \\
 \\
 \frac{\bar{a} \notin V_p[x]}{\langle \text{skipBlockHelper}(x(\bar{a})) \{ \bar{s} \}, V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \bar{s}; \text{commit}(x), V_c, V_p, D_c, D_p, \bar{x}_c + [x] \rangle} \text{ NEWVECTOR} \\
 \\
 \frac{V_c(x) = \bar{a} \quad V_p(x) = A \quad D_c(x) = A_d}{\langle \text{commit}(x), V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skip}, V_c[x := \text{null}], V_p[x := (A + \{\bar{a}\})], D_c[x := []], D_p A_d, \bar{x}_c - [x] \rangle} \text{ COMMIT} \\
 \\
 \frac{}{\langle \text{addOutputRow}(\bar{a}), V_c, V_p, D_c, D_p, [] \rangle \rightarrow \langle \text{skip}, V_c, V_p, D_c, D_p[\bar{a}], [] \rangle} \text{ OUTPUTNORMAL} \\
 \\
 \frac{\bar{x}_c.\text{last} = x_l \quad D_c(x_l) = A_d}{\langle \text{addOutputRow}(\bar{a}), V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skip}, V_c, V_p, D_c[x_l := A_d[\bar{a}]], D_p, \bar{x}_c \rangle} \text{ OUTPUTINSKIPBLOCK}
 \end{array}$$

Figure 5.3: The semantics of the skip block construct, and the effect of the skip block on the addOutputRow statement.

OUTPUTNORMAL and OUTPUTINSKIPBLOCK indicate that an output statement outside of any skip blocks (in the situation where the current skip block environment is empty) adds the new output row directly to the permanent dataset store (D_p), while an output statement within a skip block adds the new output row to the dataset buffer associated with the current skip block ($D_c(x_l)$).

Note that descendant commits can succeed even if ancestor commits fail. That is, we do not wait until all ancestor skip blocks have committed before adding a new skip block vector to V_p or new output data to D_p (see COMMIT). In general, it is not useful to think of the skip block construct as a transaction construct—it is better to think of it as a form of memoization or a way for the user to communicate which program regions relate to which output data—but the closest analogue from the nested transaction literature is the open nested transaction.

Staleness Constraints

The Figure 5.3 semantics elide the details of staleness constraints, making the assumption that users are willing to accept any match in the commit log. Here we describe the simple extensions required to support staleness constraints. Figure 5.4 shows the rules that must be altered to support staleness constraints, with t ranging over timestamps.

$$\begin{array}{c}
 \frac{V_c(x) = \bar{a} \quad V_p(x) = A \quad D_c(x) = A_d \quad \text{time.now} = t}{\langle \text{commit}(x), V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skip}, V_c[x := \text{null}], V_p[x := (A + \{\bar{a}t\})], D_c[x := []], D_p A_d, \bar{x}_c - [x] \rangle} \text{ COMMITT} \\
 \\
 \frac{\bar{a}t \in V_p[x] \quad t \geq t_l}{\langle \text{skipBlockHelper}(x(\bar{a}), t_l)\{\bar{s}\}, V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \text{skip}, V_c[x := \text{null}], V_p, D_c, D_p, \bar{x}_c \rangle} \text{ ALREADYSEENVECTOR} \\
 \\
 \frac{\nexists \bar{a}t \in V_p[x]. t \geq t_l}{\langle \text{skipBlockHelper}(x(\bar{a}), t_l)\{\bar{s}\}, V_c, V_p, D_c, D_p, \bar{x}_c \rangle \rightarrow \langle \bar{s}; \text{commit}(x), V_c, V_p, D_c, D_p, \bar{x}_c + [x] \rangle} \text{ NEWVECTOR}
 \end{array}$$

Figure 5.4: The semantics of the skip block construct extended to handle staleness constraints.

First, we adjust the commit rule to store a timestamp with each commit log record. COMMITT stores $\{\bar{a}t\}$, not just $\{\bar{a}\}$, into V_p . ALREADYSEENVECTOR indicates that to skip a block, we must now find a commit log record $\bar{a}t \in V_p[x]$ such that the associated timestamp t is greater than or equal to the staleness constraint t_l provided to the skip block. NEWVECTOR indicates that to descend into a block, there must be no commit log record $\bar{a}t \in V_p[x]$ such that the associated timestamp t is greater than or equal to the staleness constraint t_l .

Although these semantics make reference to only a single timestamp, in practice we make the same extension for both physical and logical timestamps. A logical timestamp i associated with a commit log record indicates that the associated block was executed in the i th run of the script.

Programmers can also use a few special timestamps. By default, an empty timestamp is interpreted following the semantics in the previous subsection. That is, it corresponds to using the timestamp version with timestamp $-\infty$, skipping if the block has ever been executed before. Programmers can also use a special argument to indicate that all blocks should be executed, completely ignoring the commit log. This corresponds to using timestamp ∞ .

5.4 Alternative Designs for Failure Recovery

We discuss two apparently natural but ultimately fragile alternatives to our duplicate-based recovery strategy.

URL-Based Recovery

Motivating question: Why not reload the URL that failed and start again there?

Although many sites store all necessary state information in the URL, some sites do not. We have scraped two large datasets using sites that load hundreds of logically distinct pages without ever changing the URL. We wanted a general-purpose approach to restarts, so this strategy was a bad fit for our goals.

Also, data reordering makes this approach fragile. For instance, say we want to scrape information about the top 10,000 authors in a field, according to a Google Scholar author list. The URL for the author list page includes information about the list position; however, rather than retrieving data by asking for the 400th author in the list, it asks for all authors after the current 399th author's id. If a new batch of data comes in, and author 399 sinks from 399 to 520 in the ranking, loading the same URL causes a big gap in our data. Or the author might rise in the rankings and we might be forced to resrape many authors.

In other cases, as in our Craigslist example, a given URL returns users not to a particular set of data points but to the data points now at the same position in a list. In short, even for tasks where URLs record information about the state, we would still need a duplicate detection approach for ensuring that we have reached the target data. Given that we need a same-data-check even to use a fragile URL checkpoint that can handle only a subset of scraping tasks, we chose not to center our recovery strategy on this approach.

Pre-Scraping-Based Recovery

Motivating question: Why not visit all pages of the outermost loop first, collecting the links for all items, then come back and load those links for running inner loops later?

Many webpages do store this information in actual link (anchor) tags, or in other tags that store URLs in a discoverable way. For this reason, we are interested in pursuing this technique as an optimization in future, for the cases that do store discoverable links in the DOM tree. Unfortunately, not all webpages do so. Sometimes this approach is infeasible (or slower than expected) because we need to execute a whole form interaction or some other long webpage interaction for each outer loop iteration before the desired link node is even loaded. Sometimes this is infeasible because no true link tag is *ever* present on the page, but rather a click on some other node—a div, a span—is handled by the page's JavaScript, and the JavaScript generates the target URL. As promising as this approach is for optimizing some tasks, we wanted a more general approach.

This strategy also faces the same issue that hobbles the URL-based approach. Sometimes there is no new top-level URL for loop items. In this case, the URLs we extract for loop items would be URLs for AJAX requests. Since there are often many AJAX requests associated with a given page interaction, we would first face the task of identifying which AJAX URL we should store for each loop item; then if the AJAX requests do not produce HTML, we would have to learn to parse the websites' JSON, CSV, or custom proprietary representations, rather than letting the pages' own JavaScript handle this for us.

Further, even this approach cannot completely eliminate data updating issues. Some sites get so much fresh data so often that we would still need a way to handle duplicates even with this approach.

5.5 Implementation

We discuss the details of our Helena implementation of skip blocks and briefly describe how skip blocks could be implemented in a more traditional language.

Implementing Skip Blocks in Helena

The Helena language and its associated programming environment are implemented as a Chrome extension. Although scraping is executed on the client side, the scraped data is stored on a centralized server.

To extend Helena with skip blocks, we added an additional statement to the language and an additional component to the troubleshooting section of the Chrome extension’s UI. For each loop in the program, this new UI component shows the first row of the relation associated with the loop; the first row is always observed during the demonstration phase, so this information is always available. The user can highlight columns of a relation—the key attributes of the object—then click ‘Add Annotation’ to add a skip block to the program. The tool identifies the first program point at which all key attributes are available and inserts the new skip block at that point. By default, the skip block body includes all of the remaining statements in the body of the relation’s associated loop. This placement is sufficient for our benchmarks, but the user is also permitted to adjust the boundaries of the skip blocks.

Aside from the UI design, the primary decisions that shaped our Helena skip block implementation were the choices of commit log location and format. We chose to store the commit log on a remote server along with the existing Helena data storage. The log is a table in a relational database, which makes it easy to query the log efficiently.

The full extension, including the skip block implementation, is available at <https://github.com/schasins/helena>.

Alternative Implementations of Skip Blocks

Although we have focused on using skip blocks to improve the PBD scraping experience, skip blocks can handle the same classes of problems in hand-written scraping scripts. Many modern scraping languages are implemented as libraries for general-purpose languages, which makes it easy to extend them with skip blocks. For instance, Scrapy [80], BeautifulSoup [73], and Selenium [83] can all be used as Python libraries. To introduce skip blocks to these languages, we could write our own Python library. Figure 5.5 shows part of a Python script that uses Selenium to scrape our Google Scholar running example. We have modified the real script so that rather than scraping an author’s papers directly, it wraps some of the scraping code in a `body` function; along with `[author_name, author_url]` (the key attributes of the author object), the program passes the `body` function as an argument to a `skipBlock` function. To implement the `skipBlock` function, we would simply replicate in Python the functionality that Helena implements in JavaScript. Although using a remote database for storing the commit log is still an option for this implementation, the fact that the target user is a Python programmer means that a local database is a reasonable implementation choice.

```
1 from selenium import webdriver
2 driver = webdriver.Chrome("./chromedriver")
4 # we elide most details of scraping this data; assume here we accumulate a list of author
  nodes
6 for i in range(len(author_nodes)):
7     author_name = author_nodes[i].name
8     author_url = author_nodes[i].url
9     def body():
10        driver.get(author_url)
11        paper_rows = None
12        while True:
13            old_paper_rows = paper_rows
14            paper_rows = driver.find_elements_by_xpath('//*[@id="gsc_a_b"]/tr')
15            # stopping criterion; else script keeps clicking a grayed-out next button when
              paper list ends
16            if old_paper_rows == paper_rows:
17                break
18            for row in paper_rows:
19                title = row.find_element_by_class_name("gsc_a_t").find_element_by_tag_name("a")
20                    .text
21                year = row.find_element_by_class_name("gsc_a_y").text
22                print author_name, title, year
23                next_button = driver.find_element_by_xpath('//*[@id="gsc_bpf_next"]/span/span[2]')
24                next_button.click()
24            skipBlock("Author", [author_name, author_url], [], body)
```

Figure 5.5: Part of a Python and Selenium script for scraping our running example, the Google Scholar dataset. Line 24 shows how we might use a library implementation of the skip block construct, by passing as arguments a list of key attributes and a function that encapsulates the body of the skip block.

Naturally, Python is not the only alternative host language for skip blocks. This same library-based approach extends to other languages with higher-order functions. For languages without higher-order functions, a deeper embedding (like our Helena approach) is likely a better fit.

5.6 Benchmarks

Our benchmark suite consists of seven real scraping tasks requested by scientists and social scientists from a range of fields. For each benchmark, the requester described the target data and how to navigate the website to acquire it. In most cases, we wrote and ran the Helena script (using Helena’s PBD tool); in two cases, the requester or a member of the requester’s team wrote and ran the script. In all cases, the original requesters have verified that the output data is correct and complete. To collect the benchmark suite, we asked a non-author colleague to put out a call for researchers who needed data from the web. The call went out to a mailing list of researchers from many different disciplines who all study some aspect of urban life.

Because the colleague who wrote and sent the call did not know the inner workings of our tool, we believe the benchmarks solicited from this call should not be biased towards tasks that flatter our language. This also means that each benchmark will not neatly exhibit a unique illustrative situation

Table 5.2: **Web Automation Benchmarks.** For each benchmark, we record the website of the pages that contain the target data, a description of the target data, and the discipline of the researcher who requested the data.

Dataset	Website	Data	Requester Field
Apartment listings	Craigslist	For all Seattle Craigslist apartment listings, the price, size, and other features of the apartment.	Sociology
Carpool listings	Zimride	For all rides in the Zimride University of Washington carpool ride list, the carpool listing features including start and end locations and times.	Transportation Engineering
Charitable foundation tweets	Twitter	For each foundation in a list of the top charitable foundations, the 1,000 most recent Twitter posts.	Public Policy
Community foundation features	Community Foundation Atlas	For all community foundation ons in the Community Foundation Atlas, a large set of foundation features.	Public Policy
Menu items	Yelp	For top Seattle restaurants according to Yelp, all items on the menu.	Economics
Restaurant features	Yelp	For top Seattle restaurants according to Yelp, a set of restaurant features.	Economics
Restaurant reviews	Yelp	For the top Seattle restaurants according to Yelp, all reviews.	Economics

Table 5.3: We record whether each type of duplication could be observed by each benchmark script, based on knowledge of the site’s design. We also record whether each type of duplication was actually observed in script runs reported in our evaluation.

Dataset	Can/Did exhibit unintended duplication	Can/Did exhibit site-intended duplication	Can/Did exhibit user-intended duplication
Apartment listings	Y/Y	N/N	Y/Y
Carpool listings	N/N	Y/Y	Y/Y
Charitable foundation tweets	N/N	Y/Y	Y/Y
Community foundation features	N/N	N/N	N/N
Menu items	Y/Y	Y/Y	Y/Y
Restaurant features	Y/Y	N/N	Y/N
Restaurant reviews	Y/Y	N/N	Y/Y

in the problem space. Rather, we observe the messy sets of problems that cooccur in real scraping tasks.

Table 5.2 describes each scraping task. Although our language and tool do not require that all pages in a given script be from a single website, each requested dataset used pages from only a single site. Table 5.2 includes this website, a description of the output dataset, and the discipline of the researcher who requested the data.

Table 5.3 shows a summary of our benchmarks according to whether they exhibit data duplica-

tion. We break data duplication down into two forms: unintentional and intentional. Unintentional duplication means that the website backend (which has a perfect snapshot of the data) never includes multiple instances of one data point, but the client-side view of the data does. In contrast, intentional duplication means that even with a perfect snapshot of the underlying data, the client-facing representation of the data would include multiple instances of at least one data point. For intentional duplication, we can be even more specific about who intends the duplication. In some cases, the website is designed to produce duplicates; for instance, the Zimride carpool website is designed so that a given carpool listing is associated with many rides, so the stream of upcoming rides points back to the same listing many times. In some cases, the website does not intentionally duplicate data, but users choose to post duplicate data; for instance, many users post the same apartment listings to Craigslist repeatedly. In the latter case, the website typically lacks awareness of the duplication—a different id or url may be associated with each item—but a human observer would likely consider apartment listings or restaurant reviews that share text to be duplicates.

Table 5.3 reports both whether a form of duplication is possible based on the website design and also whether we observe the duplication in the benchmark runs reported in our evaluation.

5.7 Performance Evaluation

We evaluate the effect of skip blocks on performance in a single-run context, a multi-run context, and a failure recovery context. We find that skip blocks improve performance on unstable datasets in a single-run context, on stable datasets in a multi-run context, and on most datasets in a recovery context.

Single-Run Performance

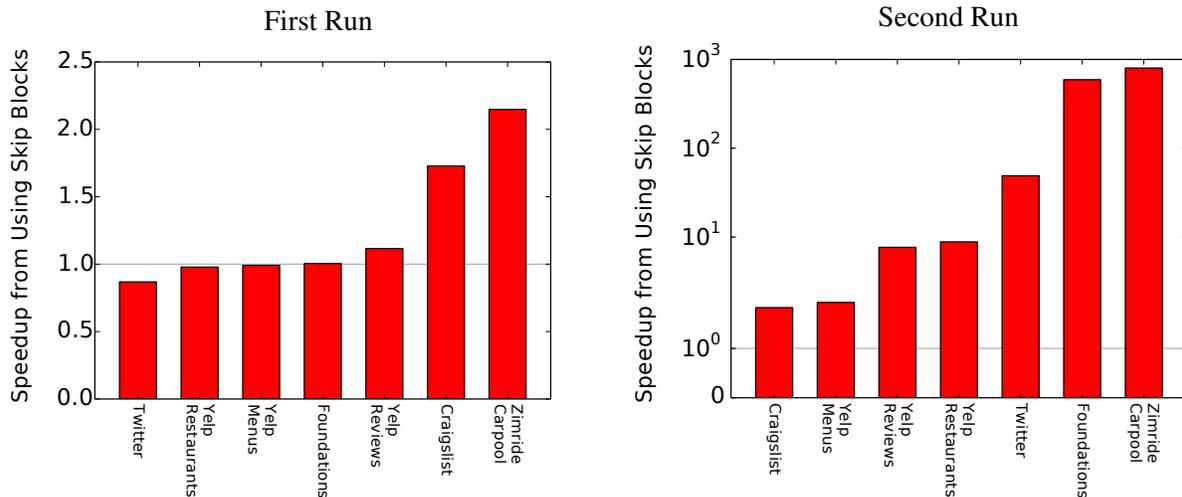
Q1: What are the performance costs and benefits of duplicate detection in a single script execution?

H1: For most tasks, duplicate detection will impose a moderate performance overhead. For tasks that involve iterating over live data or over data with intentional redundancy, it will confer a substantial performance benefit.

In practice, this means that tasks with relatively stable data (or pagination approaches that hide instability) should see little effect, while benchmarks with data that updates quickly should show substantial improvements. Even in the case of stable data, we predict performance improvements if the web interaction is designed to display a particular object multiple times.

Procedure

For each benchmark, we conducted two simultaneous runs of the script, one that used the skip block construct and one that did not.



(a) The speedup associated with using the skip block construct for the first run of each benchmark script. (b) The speedup associated with using skip block for a second run of each benchmark script one week later. Note the log scale y-axis.

Figure 5.6: The speedup associated with using the skip block construct for the first and second runs of benchmark scripts.

Results

Figure 5.6a shows the results of the single-run experiment. Speedups vary from $0.868405\times$ to $2.14739\times$, with a median of $1.00453\times$.

We give a sense of the factors at play by describing the best and worst cases for duplicate detection. At the far right, we see the best performer, the carpool listing task. Zimride allows users to post listings requesting and offering carpool rides. Each listing may offer multiple rides. For instance, a listing may repeat each week, every Monday, Wednesday, and Friday. Since many users list commutes, this is common. The Zimride web interface however, rather than offering a stream of listings, offers a stream of upcoming rides. Thus, we may see our hypothetical MWF listing three times in the first week of rides, once for the Monday ride, once for the Wednesday ride, and once for the Friday ride. However, all rides would point back to the same listing and thus be recognized as duplicates. In this case, approximately half of the listed rides were duplicates. Each time we recognize a duplicate, we avoid a network request. Since network delay accounts for most of the scraping time, the cumulative effect is to offer about a 2x speedup.

The Craigslist case is similar, except that most of the duplication comes not from intentionally displaying items at multiple points in the stream, but simply from the high volume of new listings being entered, and the fact that Craigslist’s pagination approach allows new listings to push old listings back into view.

On the other side of Figure 5.6a, we see the Twitter script offering a good example of a worst-case scenario for online duplicate detection. For the Twitter script, detecting a duplicated tweet

actually has no performance benefit. By the time we have the information necessary to check whether a tweet is a duplicate, we also have all the information necessary to scrape it. So the first-run version of this benchmark shows a case where we see all the overhead of duplicate detection with absolutely no benefit, since no important work can be skipped even if we do find a duplicate. Recall that the commit log is stored remotely, so communicating with the server to check for a duplicate before proceeding with execution can incur a substantial overhead. Future versions of the tool could certainly optimize this aspect of the interaction.

Multi-Run Performance

Q2: What are the performance costs and benefits of duplicate detection in a multi-execution context?

H2: For tasks with highly unstable data, duplicate detection will impose a moderate performance overhead. For tasks with fairly stable data, it will confer a substantial performance benefit.

In short, we predict tasks that benefit from duplicate detection in the single-execution case will not gain much additional benefit from tracking entities already scraped in previous executions. However, the highly stable datasets that receive little benefit from duplicate detection during a single run should see substantial performance improvements in later runs.

Procedure

For each benchmark, we conducted a single run using the skip block construct. After a week, we conducted two simultaneous runs of the benchmark, one that did not use the skip block construct, and one that did.

Results

Figure 5.6b shows the results of the single-run experiment. Speedups vary from $1.82723\times$ to $799.952\times$, with a median of $8.81365\times$.

Here we see how the Twitter script, the least performant of the first run, is a star of the multiple-runs scenario with a speedup of 49x. This script was designed for scraping new tweets as they come in over time, so there is no skipBlock for Twitter accounts—we want to scrape every account every time—but the script is configured to break out of the ‘for each tweet on page1’ loop as soon as it has seen 30 duplicate tweets in a row. The script scrapes up to the first 1,000 new tweets, so since most foundations tweet much less than that in a week, the duplicate-detecting script can stop after scraping only a tiny portion of what the non-duplicate-detecting script must scrape. Further, that tiny portion consists of only the most recent tweets, while the top 1,000 includes many fairly old tweets. Since Twitter loads new tweets in a given stream much, much faster than older tweets, this gives the duplicate detection version an additional advantage.

At the far left, the Craigslist speedup is about the same for the multi-run case as for the first run case. It barely benefits from records committed during the previous scrape but continues to benefit from records committed during a single scrape. This partially supports the first half of

our hypothesis, that we will see less improvement for benchmarks with frequently updated data, although we were wrong to predict a slowdown relative to the non-skip block version.

At the far right, we see Zimride with a speedup of almost 800x. This improvement reflects the fact that only one new listing was added to the Zimride site during the course of the week. (Recall that the task was to scrape carpool listings associated with a single school, and there was no semester change during the week, so this is unsurprising.) The same general explanation applies for the Foundations benchmark, which only added data for one new foundation between the two scrapes.

Failure Recovery Performance

Q3: How does our duplicate-based recovery strategy affect execution time compared to a human-written strategy that can be customized to a given script?

H3: In most cases, our duplicate-based recovery strategy will make the scrape slightly slower than the custom solutions that a programmer might write. In cases where starting from the beginning causes the script to encounter new data, duplicate-based recovery will be much slower than custom solutions.

Procedure

To evaluate the efficacy of duplicate-based recovery, we performed a limit study. We first selected three error locations for each benchmark. We selected the points at which the script should have collected one quarter, one half, and three quarters of the final dataset. For a given program point, we simultaneously ran three script variations, then simulated a failure at the target point. The three script variations are:

- A naive script: at the point of failure, this script begins again at the beginning.
- A script using skip block-based recovery: at the point of failure, this script begins at the beginning but uses the store of already-seen entities to return to the correct state.
- An ideal custom script: an imagined custom solution that can return to the correct state in zero time; in practice, we simply allow this script to ignore the simulated failure.

For each benchmark, for each error location point, we repeated this process 3 times.

Naturally, the ideal custom strategy that recovers in zero time is impossible, but the zero-time restriction is not the only restriction that makes this ideal unattainable. For some scripts in our benchmark suite, there are no constant-time recovery strategies available. We make the zero-time assumption simply to make the ideal script into the harshest possible competition. Again, this is a limit study, intended to find an upper bound on how much slower skip block recovery is than a manual, expert-written recovery strategy.

For this experiment, we used benchmark variations that collect smaller datasets than the full benchmarks used for the prior experiments. The variations were designed (based on knowledge of

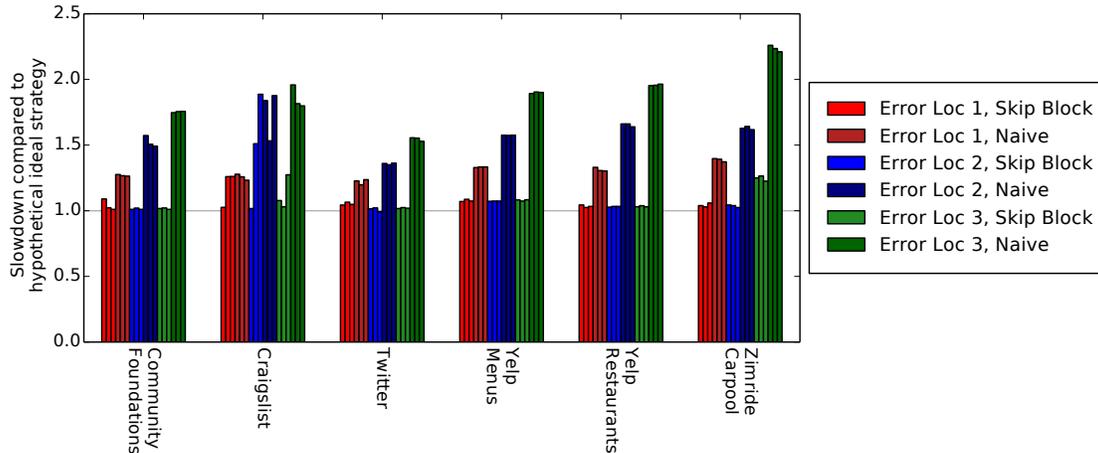


Figure 5.7: The results of the recovery strategy experiment. Each bar represents the time to complete one run, normalized by the best run time achieved by the Ideal strategy.

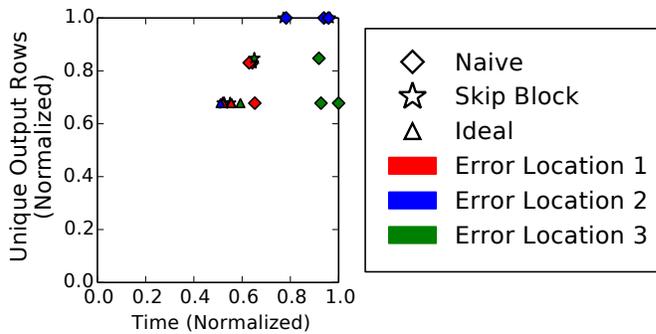


Figure 5.8: Amount of unique data collected vs. execution time for each run of the Craigslist benchmark in the recovery experiment, including runs with the hypothetical ideal recovery strategy. This reveals that Skip Block takes substantially longer than Ideal only when it collects more data.

the sites) to avoid encountering intentional or unintentional duplicates during a single execution. This allows us to disentangle the performance effects of single-execution data duplication from the effects of data updates on recovery time. As our results show, this design does *not* shelter the recovery executions from data updates.

Results

Figure 5.7 shows the results of the recovery experiment. All benchmarks except Craigslist exhibit the pattern predicted by the first part of our hypothesis. The skip-based recovery approach takes slightly more time than our hypothetical ideal. Meanwhile, the Naive approach takes about 1/4 of the ideal’s time to recover from a failure at error location 1, about 1/2 of the ideal’s time to recover from a failure at location 2, and 3/4 of the ideal’s time to recover from a failure at location 3.

The results of the Craigslist benchmark demonstrate the second half of our hypothesis, that restarting will be slower in cases where restarting from the beginning causes the script to encounter new data—that is, not only the old data already visited by the pre-error scrape. A close look at Figure 5.7 reveals that several runs of the duplicate-based strategy produce the same nearly ideal

behavior we see for other benchmarks. Others take longer; these represent the runs in which the listing relation was refreshed between the beginning of the initial scrape and beginning of the recovery scrape. If all data from that point forward is new data, duplicate-based recovery performs exactly as the Naive approach performs. If only some of the data is new data, duplicate-based recovery still outperforms Naive.

Figure 5.8 gives us more context for analyzing the recovery performance for the ‘Craigslist’ task. It reveals that in the presence of new data, duplicate-based recovery scrapes as much data as the Naive approach, but that in the absence of new data, it performs (almost) as well as Ideal. In short, even when duplicate detection makes for slow recovery, it comes with the compensating feature of additional data.

Runs that use duplicate-based recovery take 1.06658x the time of runs that use the hypothetical ideal recovery strategy. This is approximately the overhead we should expect for recovering from an error halfway through execution. Further, duplicate-based fast-forwarding is 7.95112x faster than normal execution.

5.8 Related Work

We have seen few techniques aimed at reducing the effects of data liveness and redundancy on web scraping tasks. The features whose functions most closely resemble the function of our skip block construct are the features intended to improve the performance of longitudinal scraping tasks, so we focus on these. These techniques are often called *incremental scraping* techniques. We classify works according to the skillset demanded of the user.

Tools for Programming Experts

There are many languages and libraries available for coders who are comfortable writing web automation programs from scratch. Scrapy [80], BeautifulSoup [73], and Selenium [83] are all excellent tools for various styles of web automation or web scraping, and there are many other less mainstream options [63, 28]. This category of tools and languages naturally shows the widest variety of incrementalization approaches. Many programmers use custom duplicate detection approaches to implement incremental scraping; many StackOverflow questions ask how to incrementalize individual scraping tasks, and many answers suggest approaches that suit a given webpage [85]. In general, we are not interested in these custom site-specific solutions, since they offer no guidance for a language like ours that must construct general-purpose incrementalization and recovery approaches with minimal user input.

The most general approach we have seen suggested to coders is the DeltaFetch [81] plugin for the Scrapy language [80]. For a subset of data scraping tasks, this offers developers a plug-and-play solution to incrementalizing their scrapers. However, it is not clear that the suitable subset of scraping tasks is large; for reference, only one of our seven benchmarks could be automatically incrementalized with DeltaFetch.

Tools for DOM Experts

We use ‘DOM expert’ to refer to users who, although they are not comfortable writing code, understand XPath, how templates are used to generate webpages, how JavaScript interacts with DOM trees, and other web internals.

There are few tools in this category; it seems tool designers assume that if users can understand the intricacies of the DOM and other web internals, they must be able to or prefer to program. However, we have found one tool that mostly fits into this category, although many of the more advanced features require programming ability. Visual Web Ripper is a tool or visual language targeted at DOM experts [87]. It offers a “duplicateAction” that can be attached to a page template to indicate that the page template may produce duplicates, and it gives the user the option to remove the duplicate data or to cancel the rest of the scrape. Users also have the option to provide a Java function that can be run to decide whether it is a true duplicate and thus whether to cancel the page template. In addition to the fact that the tool is inaccessible to non-DOM experts, their cancellation options are a much weaker form of control than our skip block; also, the fuller control available with the Java function is only accessible to programmers. This mechanism is certainly related to ours in that it is based on detecting duplicates, but while our skip block approach could be used to implement the options Visual Web Ripper provides, the reverse is not possible.

Tools for End Users

Prior to our own work, we know of no incrementalization feature targeted at end users. However, this is not primarily because of incrementalization itself, but rather seems to be a function of the web scraping tools that are available to end users. We consider Helena to be a full-featured web automation language, which makes it capable of expressing the kinds of scripts that benefit from incremental scraping. In contrast, most prior end-user scraping tools offer very limited programming models. They typically fall into one of two categories: (i) record and replay tools or (ii) relation extractors.

A web record and replay tool takes as input a recording of a browser interaction and produces as output a script for replaying that same interaction on fresh versions of the pages. Ringer [9], Selenium Record and Playback [82], CoScripter [51], and iMacros [33] all provide this functionality, as do many others [39, 54, 30, 52, 69, 22], although some typically require debugging by a programmer or DOM expert and thus might be better treated in the categories above. Since a replayer is only responsible for replaying a single interaction, there is no duplication within a single run, and even incrementalizing could offer only the barest performance benefit, so no current replay tool offers a built-in mechanism for incremental scraping.

A web relation extractor takes as input a webpage and a subset of the webpage’s DOM nodes, labeled with their positions in a relation. Its output is a program that extracts a relation of DOM nodes from the webpage. Many tools from the wrapper induction community offer this functionality [19, 13, 44, 61, 91, 42, 66, 20, 3]. More recently, FlashExtract [50], KimonoLabs [37], and import.io [32] have offered highly accessible tools with this functionality. This form limits these tools to scraping all cells in a given row from a single page. Since preventing unnecessary page loads is the

primary mechanism by which incrementalization improves performance, there is little incentive for relation extractors to offer incremental scraping.

Although both replay and relation extraction are natural building blocks for end-user-friendly languages, they are not themselves full-fledged web automation languages. None of the tools listed above can scrape any of the datasets in our benchmark suite. So it is natural that they have not had to tackle incremental scraping.

Of all the fully end-user-accessible web programming tools we know, only one brings together both replay and relation extraction. This is the Vegemite [53] tool, mentioned in more detail in Chapter 3. Vegemite is essentially a tool for extending an existing relation with additional columns. The authors use the example of having a table of houses and their addresses, then pasting the address into a walk score calculator and adding the walk score as a new column. This is a somewhat restrictive programming model, since all programs must share a single simple structure. (Of the seven benchmarks in our suite, only the Community Foundations Atlas script can be expressed with this program structure.) So again, this falls short of being a full-scale web programming language, but it certainly brings us closer. The creators of the Vegemite tool never chose to tackle incrementality, presumably because the user provides the input table and is presumed to have done any duplicate detection as a pre-processing stage. Again, in this context, there is little benefit to online duplicate detection.

5.9 Conclusion

This chapter introduces a novel construct for hierarchical memoization of web scraping executions with open transaction semantics that is accessible to end users. Our skip blocks improve web script performance by posing users a simple question that, as our user study reveals, end users can answer well and quickly. With recovery speedups of 7.9x, the average user with a poor network connection can run much longer-running scripts than they could feasibly run without skip blocks. Scripts collect more data in less time, and they are less derailed by the inevitable server and network failures. This language feature brings larger datasets in range and makes recovery fast enough that users can easily interrupt long-running scripts when it comes time to close the laptop and head home. Combined with the benefits for handling live data and redundant web interfaces, and given that it offers incremental data scraping by default, we see this as an important step towards developing a highly usable web automation language.

Chapter 6

Skip Blocks for Parallel and Distributed Execution

Many web scrapers run for hours or days and can benefit from parallelization. Modern web scrapers like Helena emulate the user rather than the browser—rather than sending requests directly to the web server, they issue commands to the browser UI. These *user-emulating* web scrapers face a parallelization problem with two defining characteristics: (i) Non-Transferable Work: Parallel workers cannot assign or steal units of work because websites often do not expose an efficient handle for each unit of work—URLs are efficient handles, but websites are moving away from exposing intermediate state in URLs. (ii) Inconsistent Workloads: Web servers often serve different data to different clients, or to the same client at different points in time, so parallel browsers may receive differing workloads. We describe this *transferless parallelism* problem, discuss applicable parallelization strategies, and empirically evaluate two parallel algorithms on real-world, large-scale web scraping tasks.

6.1 Introduction

As the demand for web data rises, the demand for large-scale data also rises. Parallelization has the potential to drastically accelerate scraper execution by hiding network latency, which is the primary driver of scraper run times. With large-scale scrapers running for hours or days, the value of parallelization in this domain is clear.

User-Emulating Web Scrapers

There are two styles of web scrapers, browser-emulating (BE) and user-emulating (UE). A BE scraper bypasses the web browser and accesses the web directly using the Hypertext Transfer Protocol, issuing requests for URLs, then receiving and parsing the responses. The requests can include standard top-level URLs (the URLs a browser user types into the URL bar, which should return a webpage) and AJAX URLs (the URLs a webpage’s JavaScript uses internally, which can

return data in any format). In contrast, a UE scraper accesses the web by interacting with a web browser, dispatching user-like actions such as clicks and keypresses. In UE scraping, the browser and the webpages' own JavaScript code request URLs and parse the responses.

We focus in particular on UE scrapers. Programmers write UE scrapers for a variety of reasons, but especially: (i) They lack the browser expertise to reverse engineer browser-server communication but have enough programming skills to use a language like Selenium [83] to automate user-like actions, like clicks. Many data scientists and social scientists fall into this category. (ii) Programmer or non-programmer, they write scrapers with demonstration-based authoring tools that synthesize UE programs—e.g., Vegemite [53], Helena. (iii) To circumvent websites' bot-targeted rate-limiting behaviors. (iv) To scrape websites that do not produce unique URLs for logically distinct pages (altering a webpage with AJAX-loaded content does not update the displayed URL by default—the webpage designer must choose to update it) or websites that make request URLs or parameters undiscoverable except via UI manipulations (typically by generating the URLs and parameters in JavaScript in response to UI actions, rather than storing them in an HTML document).

Naturally, UE scrapers represent only a subset of the broader range of web scrapers, so the parallelization problem we describe in this chapter does not apply to all web scrapers. However, the factors that drive the choice of user-emulating strategies are on the rise. (i) Increasing numbers of novice programmers and data scientists are becoming interested in web data [6, 56, 89]. (ii) More demonstration-based tools are becoming available [53, 37, 32, 15]. (iii) Websites are increasing their efforts to detect bots [5, 1, 75]. (iv) Finally, websites are becoming increasingly interactive and AJAX-intensive [71, 23, 59]. In short, the UE style of scraper is sufficiently prevalent that we feel characterizing their unusual parallelization problem is worthwhile.

Transferless Parallelism

User-emulating web scrapers face a parallelization problem we call *transferless* parallelism. Transferless parallelism is defined by:

Non-Transferable Work: We cannot transfer units of work between parallel workers. Traditionally, web scrapers have used URLs as pointers to units of work. Unfortunately, this approach does not align well with modern web design trends. Over the past decade, the rise of the Single Page Application (SPA) has drastically increased the amount of webpage content that is loaded via JavaScript rather than in response to an initial URL [59, 71, 23]. Rather than having a set of URLs that each draw down a full webpage, many modern sites load a 'single page' that includes logic for retrieving small amounts of information from the server (via AJAX requests) and formatting it in the single page. The goal is to offer better interactivity and faster loading times. Modern websites thus subvert the URL-pointer approach by: (i) failing to generate new URLs for logically distinct pages (with AJAX-loaded content) or (ii) generating URLs for AJAX-adjusted pages dynamically, which makes the URLs difficult to scrape without first paying the performance price of loading the page. Without direct URL pointers to all units of work, the only general-purpose pointers are clones of a web browser and sequences of UI interactions that start at a root URL and manipulate the browser to reach the target data. In short, there are no efficient, general-purpose pointers for

accessing units of work, whether to transfer them between workers or even to save work for later processing.

Inconsistent Workloads: Parallel workers receive different input data. Units of work cannot be transferred between workers (see Non-Transferable work above), but they *can* be transferred from the original source, the web server, to a parallel worker. Thus, workers can discover the structure and content of the scraping workload by starting at an initial webpage and navigating to other logical pages that are accessible via links or other interactions. However, web servers may choose to display different underlying datasets to different parallel workers. This is particularly common on webpages that display user-generated content or other frequently updated content, but it also occurs in a vast array of other scenarios, including when websites engage in A/B testing, when they re-rank ordered lists frequently, or when they adapt content to respond to changing beliefs about user interests and preferences. The end result is that parallel workers can see different input workloads. Even a single worker may see inconsistent reads of the underlying data store since the contents of different page loads during one execution are not drawn from a single atomic read of the data store. In short, an external actor, the web server, which is not under the control of the parallel scraping algorithm, is modifying the input data throughout execution of the scraping algorithm.

These characteristics make traditional parallel patterns inapplicable. The non-transferable work constraint implies:

- We cannot efficiently move tasks from one browser to another. → We cannot use work sharing, work stealing, or fork-join patterns.
- We cannot store tasks for later execution. → We must choose to execute or ignore tasks upon finding them.

The inconsistent workload constraint implies:

- Input trees are not known statically. → We must assign tasks to workers at runtime.
- Workers receive different input trees. → n workers cannot, for instance, claim every n^{th} task they encounter.

This chapter explores how to design parallel scraping algorithms in the face of these unusual constraints.

Contributions

This chapter contributes:

- A characterization of the *transferless* parallelization problem that user-emulating web scrapers face.
- Signature-based algorithms for parallelizing user-emulating scrapers, one that prioritizes load balancing and one that prioritizes limiting communication among workers. Both work with an off-the-shelf web browser.

- An empirical evaluation of how our signature-based parallelization strategies perform on real, large-scale web scraping tasks. We achieve up to $7\times$ speedup even on a desktop, and in a distributed setting we can reduce 50 minute tasks to 42 seconds.

6.2 Background

We start with a working example, a sample of the kinds of UE scrapers that face the transferless parallelism problem. The following pseudocode program loads a Yelp top-1,000 restaurants list and iterates through the restaurants then, for each restaurant, through a list of restaurant reviews:

```

1 restaurant_dataset = freshDataset() // the output datasets
2 review_dataset = freshDataset()
3
4 p1 = load("https://www.yelp.com/<city_name>")
5 type(p1, searchBarNodeIdentifier, "restaurants") // type in search bar
6 p2 = click(p1, searchButtonNodeIdentifier) // click on search button
7 // iterate through restaurants in search results
8 for name, address, avg_rating in p2.restaurants{
9   p3 = click(p2, name)
10  r = restaurant_dataset.addRow([name, address])
11  for reviewer_name, rating, review_text, review_date in p3.reviews{
12    rev = review_dataset.addRow(r, [reviewer_name, rating, review_text])
13  }
14 }
```

Variables `p1` through `p3` refer to webpages. The expression `p2.restaurants` in line 5 evaluates to a reference to the restaurants relation represented in page `p2`. The function `restaurants` maps a page to a table of restaurant data in the page. The loops hide details of using ‘Next’ buttons to access additional pages of restaurant data or review data.

We use this example to illustrate how web servers can change the input data trees they serve to parallel workers, in this case the tree of restaurants and their reviews. Table 6.1 describes which features of the input data tree are guaranteed to be stable across workers and which are not. Our Yelp task scrapes the top 1,000 restaurants in a city according to Yelp’s proprietary ranking algorithm. Empirically, Yelp shows different rankings to different parallel workers and in fact shows different sets of restaurants in the top-1,000 list to different worker processes, whether for A/B testing or another internal purpose. Thus the **order** and **set** of child nodes can be inconsistent.

Content changes naturally lead to size and shape changes. If a restaurant with few reviews is removed from the top-1,000 list and a restaurant with many reviews is added, the shape of the tree changes, and the size of the tree increases. Likewise, if a new review is posted between the time

Stable across workers	<ul style="list-style-type: none"> • The maximum height of the tree.
Not stable across workers	<ul style="list-style-type: none"> • The order of the child nodes of a given node. • The set of child nodes of a given node. • The number of child nodes of a given node.

Table 6.1: **Stable and Unstable Features of Web Automation Workloads.** A summary of input data tree features that are stable and unstable across parallel workers.

when worker 1 and worker 2 load a restaurant’s page, the size of the restaurant’s subtree is different across processes—the **number** of child nodes varies.

The assignment of subtasks to workers is the core challenge of parallelizing UE scrapers. Assignment techniques must balance many competing needs: the need to collect all of the target data, to avoid collecting multiple copies of the target data, to offer good load balancing, to avoid re-executing large amounts of work to reach a target task. The inconsistency of data trees described in this section substantially constrains the assignment problem. We cannot assign subtasks to parallel workers statically when we do not know the set of tasks statically. Assigning the first of n workers to handle the first k/n tasks is impossible since k , the number of tasks, is unknown. Likewise, we should not instruct each of the n workers to take every n^{th} task, because each process may see a different ordering of the subtasks. The fact that different tasks at the same level of the hierarchy can have different sizes also suggests the potential value of dynamic load balancing in this domain.

6.3 Signature-Based Parallelization

This section describes signature-based parallelization, a class of techniques for handling transfer-less parallelism. The key idea is to associate independent subtasks with programmer-provided signatures—essentially unique IDs for the subtasks, which workers can extract or construct when they encounter a new subtask. Each parallel worker then traverses its own input data tree, using signatures to determine whether to execute or ignore each subtask.

The programmer chooses subtask signatures and associates each signature with the block of code that executes the subtask. The programmer must assign signatures such that: (i) Whenever any worker finds a subtask that processes a given logical node of the input data tree, the subtask produces the same signature. (ii) Two subtasks produce the same signature only if they process the same logical node of the input data tree. (iii) Obtaining a signature for a subtask is more efficient than executing the subtask.

For the web scraping domain, object attributes offer readily available signatures. For instance, in our restaurants and reviews example, if we iterate through a list of restaurants that includes the name, address, genre, and neighborhood of each restaurant, the signature for a restaurant could be its name and address. The associated block of code would be the code that follows a restaurant link and collects its reviews. This meets our three criteria: (i) If workers encounter a given restaurant in their restaurant lists, the restaurant will always have the same name and address and thus will always produce the same signature. (ii) If a worker finds a restaurant with the same name and address as another restaurant, we can safely assume it is the same restaurant. (iii) Extracting the name and address of a restaurant from a list that includes names and addresses is faster than following the restaurant’s link, loading a list of reviews, and iterating through all reviews.

Once we have a signature associated with each independent subtask, the remaining design challenge is to assign subtasks to parallel workers based on signatures. The goal is to design the assignment process such that:

- If the input data tree is the same across all parallel workers and throughout time and each data node appears exactly once in the data tree, each data node appears exactly once in the output data.
- The assignment strategy offers good load balancing.
- The assignment strategy makes it possible to recover from network, server, or worker failures.

In this section, we describe hash- and lock-based strategies.

Hash-Based Task Assignment

```

1 Procedure worker.execute(signature, block, Vi)
2    $h \leftarrow \text{hash}(\text{signature})$ 
3   if  $h \in V_i$  then
4      $\text{block.execute}()$ 
5   end

```

Algorithm 1: Hash-Based Assignment

We first consider a hash-based, communication-free approach for assigning subtasks to workers. This strategy follows the general approach laid out in Algorithm 1. The core idea is that the signature for any given object can be hashed (line 2), and each worker w_i is responsible for a partition V_i of all hash values. In this scheme, a given worker executes the block for an object if and only if the object’s signature hashes to a value in the worker’s hash values (line 3).

Output Data Correctness

If all workers see the same stable data tree throughout a run, each data node will be mapped to exactly one worker. Because the data is stable, a node’s signature remains stable throughout and thus hashes to a stable value, which is associated with a single worker, w_i . Thus, w_i executes the block for the node, all other workers pass over the block for the node, and the node appears exactly once in the output data.

Note that if the node appears multiple times in the stable data tree, w_i executes the block each time and adds its data to the output dataset each time. For any given task, this may or may not be desirable behavior. Our specification—for stable data in which each logical node appears exactly once, the node’s associated block should be executed exactly once—does not constrain behavior in this case.

Load Balancing

Overall, load balancing of a naive hash-based variation is poor. (See Section [6.4](#).) In the web scraping domain, nodes of the data tree, even if they appear at the same depth, often require dramatically different amounts of processing time. A naive approach that partitions hash values

evenly across workers exhibits good load balancing if the node signatures produce uniformly distributed hashes and the amount of time to process each node is approximately the same. Naturally, more sophisticated hash-based designs can improve load balancing behavior; for instance, we can allow workers to transfer control of subsets of their assigned hash values.

Failure Recovery

When a worker experiences a network or server failure in a hash-based scheme, we can recover by starting a new worker, assigning it the same partition of hash values that the failed process controlled, then executing the scraping program from the beginning. If the implementation commits blocks and their associated output data during execution, the recovery worker can consult the commit log to avoid re-executing blocks completed by the disrupted worker. Otherwise, the recovery worker must re-execute the blocks completed by the disrupted worker; this affects performance but does not compromise the output data's correctness.

Lock-Based Task Assignment

```

1 Procedure worker.execute(signature, block)
2   if coordinator.execute?(signature) then
3     block.execute()
4     coordinator.commit(signature)
5   end

6 Procedure coordinator.execute?(signature)
7   execute ← true
8   lock (lockLog[signature], commitLog[signature])
9   if signature ∈ lockLog then
10    | execute ← false
11  end
12  if signature ∈ commitLog then
13    | execute ← false
14  end
15  if execute then
16    | lockLog.insert(signature)
17  end
18  unlock (lockLog[signature], commitLog[signature])
19  return execute

20 Procedure coordinator.commit(signature)
21  lock (lockLog[signature], commitLog[signature])
22  commitLog.insert(signature)
23  lockLog.remove(signature)
24  unlock (lockLog[signature], commitLog[signature])

```

Algorithm 2: Lock-Based Assignment

A lock-based strategy for assigning subtasks to workers allows each worker to claim a node if and only if it is the first worker to reach the node. This strategy follows the procedure in Algorithm 2 and relies on a coordinator process. Upon encountering a data tree node, a worker checks for a lock on the node's signature before attempting to execute the block (line 9), then executes the block only if it acquires the lock before any other worker (lines 2 and 16). Once the worker processes the node, the coordinator releases the lock (line 4 and 23) but adds it to a commit log (line 22), so no other workers can claim the node (line 12).

This approach is flexible, with variations on when locks are acquired and lock exclusivity defining a space of lock-based implementations. In particular, programs can include nested blocks, and there are multiple ways to choose the level at which locking should occur. We briefly describe three points in the space to suggest the design considerations.

Outermost Locking. For this simple locking strategy, workers only claim locks on the outermost block in a program, and workers cannot share locks. For a program scraping restaurants and reviews, workers acquire locks for restaurants, and once a worker claims a restaurant, no other worker can enter the block associated with that restaurant.

Adaptive-Granularity Locking (Counting Semaphores). For a task like our restaurant example with 1,000 top-level subtasks, the outermost locking approach typically load balances well, even though some workers may sit idle at the end. However, if we want to use 10 workers to scrape only the top 5 restaurants, 5 processes will claim restaurants, and the remaining 5 processes will sit idle. In an adaptive-granularity approach, all blocks, including nested blocks, lock their subtasks. When a worker finds no more unclaimed objects for the outermost block, this strategy converts the locks associated with the outermost block from binary semaphores to counting semaphores with limit 2, and a worker that runs out of tasks executes the program again from the beginning, allowed to enter any locked task with a semaphore under 2; if the limit is at 2 and all locked tasks have their semaphores at 2, the limit rises to 3, and so on. The limit for locks associated with all inner blocks remains at 1 until we run out of subtasks at that level and all levels above it, at which point we begin the same limit-raising procedure.

In this scheme, our 5 restaurants and 10 workers run is no longer poorly load-balanced. The first 5 restaurants are claimed by the first 5 processes. The next 5 processes reach the end of the loop without acquiring any tasks, so the limit is raised to 2. Processes 5 through 10 begin already-claimed restaurant tasks, such that each restaurant is handled by two processes at once (assuming all 5 restaurants are still being processed). At this point, workers can claim tasks at the second level. The worker joining restaurant 1 passes over any reviews already committed (or locked) by the first worker, and the workers move forward with a new review only if they can acquire the relevant lock. If one of the restaurants is completed before others, the restaurant-level semaphore limit is raised to 3, and the newly idle processes help with the remaining restaurants. If all processes are working on a single remaining restaurant and there are no more unclaimed reviews, the semaphore for review subtasks is incremented, and workers start sharing reviews as well as restaurants.

Adaptive-Granularity Locking (Greedy). Although the counting semaphores approach described above works well for *stable* data trees, this is not a good general-purpose approach for real web data. Recall that the children of a given node may vary across worker processes or over time. This means a subtask that has been locked by one worker may not be accessible to another. In our 5 restaurants example, say 5 workers were the first to claim a restaurant, the counting semaphore limit was raised to 2, and 4 additional workers joined 4 of the locked restaurants. However, consider the case where the restaurant that has only one worker does not appear in the final worker's list of restaurants. The threshold for the counting semaphores is not raised, because one locked restaurant still has a semaphore at 1, so the final worker sits idle, even if the worker *does* have access to the other four restaurants. In the worst case, no workers are able to find the single-worker restaurant, and all workers sit idle as they finish their other tasks.

A more general-purpose strategy handles changing web data by using a greedy approach. As soon as an idle worker finds any node that has been locked but not yet committed, it enters the node's subtree and begins processing the nested subtasks. This can result in more workers processing some subtrees than others, but it never results in a worker sitting idle if it has access to any remaining nodes. In this scheme, if a worker reaches the end of the execution without finding any nodes to complete with the current block granularity—e.g., if only depth-0 locks are currently multi-claimable and the worker iterated through all depth-0 nodes without finding any depth-1 tasks to claim—it starts the script again at the beginning, but treating the next level of the block hierarchy as multi-claimable. Although we may achieve marginally better load balancing with the semaphore-based approach in stable-data scenarios, this greedy technique is a more general-purpose solution for handling both stable and unstable data.

Output Data Correctness

If all workers see the same stable data tree throughout execution, these locking-based strategies process each logical node exactly once. Note that in contrast to hash-based strategies, locking-based strategies process each node exactly once even if the node appears at multiple locations in the data tree. (The commit log that prevents other workers from processing a node that has already been processed by worker w_i also prevents worker w_i from processing the node again, while the hash-based approach allowed w_i to process the same node each time it hashed to V_i .) Thus, although both approaches meet our correct output data criterion, they can produce different output datasets even in the case of stable input data tree.

Load Balancing

The quality of lock-based strategies' load balancing depends on the specific design. Overall, lock-based approaches produce good load balancing for many real web automation tasks. (See Section 6.4.) Even the simple approach taken by the outermost locking design produces good load balancing unless there are few subtasks, or unless a few subtasks require much more time than others and dominate the run time. For a more sophisticated locking approach, like our adaptive-granularity design, load balancing is even better.

Failure Recovery

When a worker fails in a lock-based approach, it leaves one (or more, if there are nested blocks) locks claimed but not committed. To recover, we delete from the lock log any signatures that the disrupted worker added to the lock log but not the commit log. A recovery worker executes the scraping script from the beginning.

Communication Across Processes

Neither the lock- nor hash-based approach relies on shared memory or fast local communication among workers. Thus, both approaches can be cleanly adapted to a distributed setting by starting worker processes on multiple machines. This has the interesting side effect of making these techniques usable even on sites that use rate limiting or trigger CAPTCHAs if they receive too much traffic from a single IP address.

6.4 Evaluation

Our evaluation aims to answer the following questions:

- RQ1: What is the effect of parallelization on output datasets? (Section 6.4)
- RQ2: How well do hash-based approaches scale for parallel web scraping? (Section 6.4)
- RQ3: How well do lock-based approaches scale for parallel web scraping? (Section 6.4)
- RQ4: Can we use distributed execution to accelerate tasks that do not scale well with parallel execution? (Section 6.4)
- RQ5: How does our generic parallelization strategy compare against hand-coded alternatives? (Section 6.4, a case study.)

All parallel experiments were performed on a Dell OptiPlex 9020 with a 3.60GHz Intel Core i7-4790 (4 cores, 8 threads) and 32GB 1600 MHz DDR3. All distributed experiments were performed on Amazon EC2 t2.micro instances with an Intel Xeon family processor up to 3.3 GHz (1 vCPU) and 1GB memory.

Implementation

To evaluate the performance of the signature-based approaches described in Section 6.3 for real large-scale web scraping tasks, we implemented three parallelizing runtimes for the Helena web automation language. The implementations are available at <https://github.com/ericniebler/helena>. Although we implemented signature-based parallelization for Helena, the implementation strategy would be the same for other UI-manipulating languages such as Selenium. To provide unique task signatures for blocks of code, the programmer picks a set of object attributes—e.g., the

Table 6.2: A suite of long-running web automation tasks requested by social scientists and data scientists.

Dataset	Data
Craigslist Listings	For Craigslist apartment listings, the price, size, and other features.
Zimride Listings	For all rides in a Zimride carpool ride list, the carpool features including start and end locations and times.
Twitter Tweets	For each foundation in a list of the top charitable foundations, the 1,000 most recent Twitter posts.
Community Foundations	For all community foundations in the Community Foundation Atlas, a large set of foundation features.
Yelp Menu Items	For city's top restaurants according to Yelp, all items on the menu.
Yelp Restaurant Features	For city's top restaurants according to Yelp, a set of restaurant features.
Yelp Restaurant Reviews	For city's top restaurants according to Yelp, all reviews.

name and address of a restaurant, the title and year of a paper. In the Helena program below, the body of the `task` construct, the block that scrapes restaurant and review information, is associated with a signature composed of a restaurant's name and address. Thus any restaurant nodes that have the same name *and* address should be treated as the same node.

```

1 // ... reach restaurants list
2 for name, address, avg_rating in p2.restaurants{
3     task(Restaurant(name, address)){
4         p3 = click(p2, name)
5         // ... scrape restaurant and its reviews
6     }
7 }
```

To execute programs that use the task construct, one of our parallelizing runtimes uses the hash-based approach described in Section 6.3 with equisized hash value partitions. The others use the lock-based approach described in Section 5; one uses the simple outermost locking design, and the last uses the greedy adaptive-granularity design. All runtimes, including the hash-based runtime, use a commit log to avoid re-executing work that has already been executed; this causes the hash-based and lock-based strategies to produce the same outputs for stable datasets, even if a given data node appears multiple times in the data tree. (See Section 24 for a discussion of why hash- and lock-based strategies can produce different outputs in this case.)

Our hash-based runtime uses a slight variation on Dan Bernstein's djb2 hash [10]. Our lock-based runtimes use a remote, centralized database to store a table of lock records.

Benchmark Suite

We use the suite of long-running web automation tasks introduced in Section 5.6. Table 6.2 describes the benchmarks. Each benchmark task involves scraping a dataset requested by a social scientist or data scientist. For example, 'Twitter Tweets' is the task of scraping the most recent 1,000 tweets by each of 50 charitable foundations.

Effect of Parallelization on Output Data

Hypothesis. We expect that parallelization should increase the amount of data a scraper execution collects in cases where data nodes expire.

Procedure We ran each benchmark with 1, 2, 4, 6, and 8 workers, using each of our parallelizing runtimes, and observed the sizes and content of the output datasets.

Strategy-Independent Trends

Increased Data Via Expiration. Runs of the ‘Craigslist Listings’ task were affected by two competing forces: (i) adding workers leads to *faster execution*, and (ii) faster execution leads to seeing more non-duplicate posts before the end of the list, which leads to *slower execution*. We see (ii) because Craigslist paginates by indexing into the master list when a user presses the ‘Next’ button. This means that if x new ads are posted between loading page 1 and loading page 2, the last x ads from page 1 appear at the top of page 2; our commit log means we pass over these duplicate ads, but the other effect of adding x new ads to the head is that the last x ads from the master list are no longer accessible. (Craigslist truncates results pages at 2,400 results, so any ad after that threshold ceases to be accessible from the results page.) Thus, if x new listings are added per hour, and our scraping execution time goes from two hours to one hour because of adding an additional worker, x of the 2,400 slots that would have been duplicates in a single-worker run are instead filled with non-duplicate, unexpired data in the multi-worker run. In short, a multi-worker run sees more data. Because new listings are added to Craigslist so quickly, slower runs in fact saw substantially less data. Concretely, in the first 240 listing slots (the first two pages), a single-worker run typically saw about 121 unique listings, while a 6-worker run typically saw about 214 unique listings. Since listing that are pushed beyond the 2,400 threshold essentially expire, this supports our initial hypothesis.

Increased Data Via Inconsistent Input Trees. The Craigs-list benchmark scrapes expiring data and exhibited the strongest effect of (ii), but surprisingly several benchmarks see more data when running more workers. For instance, Yelp showed different restaurant rankings to different worker processes—not just shuffled variations of the same restaurant set, but different sets of restaurants in the top 1,000. Concretely, imagine Yelp lists only the top 10 restaurants—then a sequential version observes exactly 10 restaurants, while a version with 8 workers could observe as few as 10 or as many as 80 restaurants. In practice, the difference is not this drastic, but we saw cases where multi-worker Yelp executions observed 11% more restaurants than single-worker executions because of re-rankings. In short, inconsistent input data across parallel workers also increases the amount of output data collected.

Hash- and Lock-Based Outputs

Adding a commit log for the hash-based implementation causes the output data of the hash- and lock-based implementations to be consistent if data is stable, but we do not get this guarantee if different workers see different data trees.

Recall that we consider two nodes equal if and only if all their signature attributes are the same. Let us use Node A to refer to a particular object that a website’s underlying data store treats as a single (mutable) object, but that a programmer’s attribute signature may treat as multiple data points.

Multiple hash-based workers may execute a block associated with Node A because an attribute differs across different page loads, producing different hashes. This is similar to the behavior of a lock-based run; each Node A variant would produce a different signature and thus a different lock, so Node A could be processed multiple times by either strategy. However, in a hash-based scheme, we can observe a case where all workers see Node A, with varying attributes, but no worker executes the associated block, because each finds that it hashes to a different worker. In this instance, the behavior of the lock-based and hash-based variations substantially changes the output. For the same situation, the lock-based strategy would scrape all new-attribute versions of Node A that are observed, while the hash-based strategy may scrape zero or some or all.

Hash- and lock-based strategies can produce different output datasets even if the attributes of all objects are stable. For instance, assume we want to scrape only the first 10 restaurants in ‘Yelp Restaurant Features,’ but we use two workers, and worker 1 loads a list that has no restaurants in common with worker 2’s list. The lock-based version will scrape all 20 restaurants. The hash-based will scrape 0 to 20—on expectation, approximately half of each page’s restaurants, for a total of about 10 restaurants. In general, the lock-based strategy will tend to collect more data than the hash-based strategy. Empirically, for the Yelp restaurants case, the lock-based runtimes scraped 8.6% more restaurants than the hash-based runtime. For our benchmark suite, the highest-divergence benchmark scraped 10.1% more leaf nodes with the lock-based than the hash-based runtime.

However, when all object attributes and the list of objects presented in the webpages is stable during the execution and no two nodes in the data tree share a signature, the lock-based and hash-based approaches both scrape each object exactly once. Thus, for static datasets with no duplicate nodes, both strategies produce the same output data.

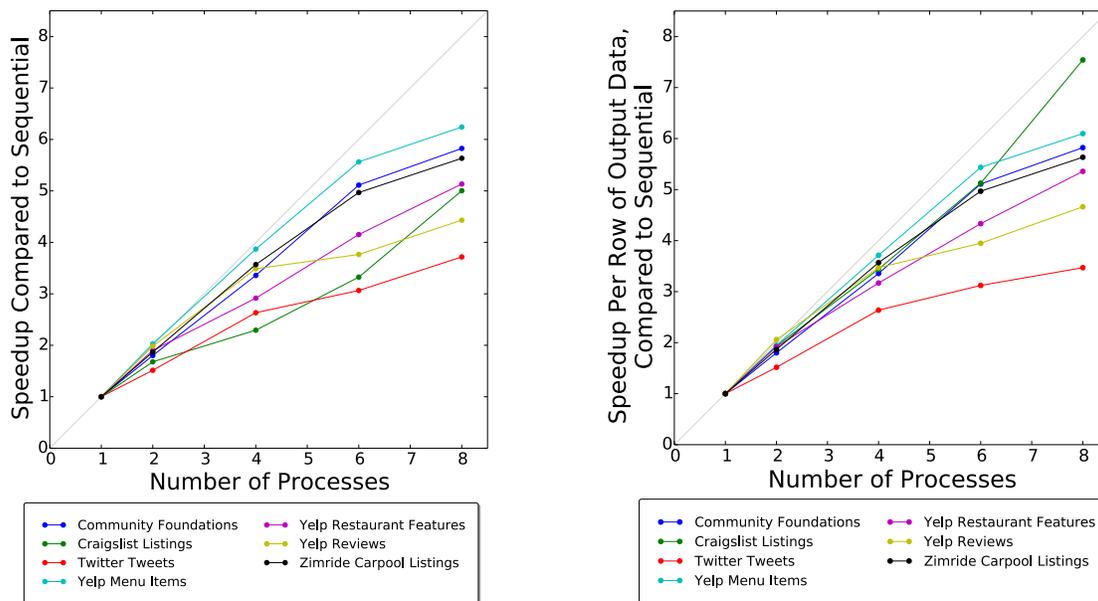
Hash-Based Runtime

Hypothesis. We expect that our hash-based approach should scale well for tasks with large numbers of subtasks of approximately equal size.

Procedure We evaluated the hash-based signature technique by running each benchmark with 1, 2, 4, 6, and 8 workers. Figure 6.1a shows the execution times for all benchmark-workers pairs, normalized by the execution time for the sequential variation.

Results. The results in Figure 6.1a reveal that the performance for the hash-based strategy varies by benchmark. Overall, this is the expected result because hashing with equisized, non-adapting hash value partitions relies on the assumption that each block encompasses a similarly sized amount of work. For some tasks—e.g., ‘Yelp Menu Items,’ ‘Community Foundations,’ and ‘Zimride Carpool Listings’—this assumptions held, and indeed we observed good performance for those benchmarks.

Other benchmarks exhibit worse performance, but not always because individual blocks encompassed differing amounts of work. In particular, the ‘Craigslist Listings’ task was affected by the expiring data phenomenon described in Section 6.4. Executions with more workers collected more data and thus took longer. To adjust for the fact that varying the number of workers may also vary



(a) **Time (Parallel Execution).** Run time for benchmark executions, normalized by serial run times.

(b) **Effective Speed (Parallel Execution).** Run time per row of output data, normalized.

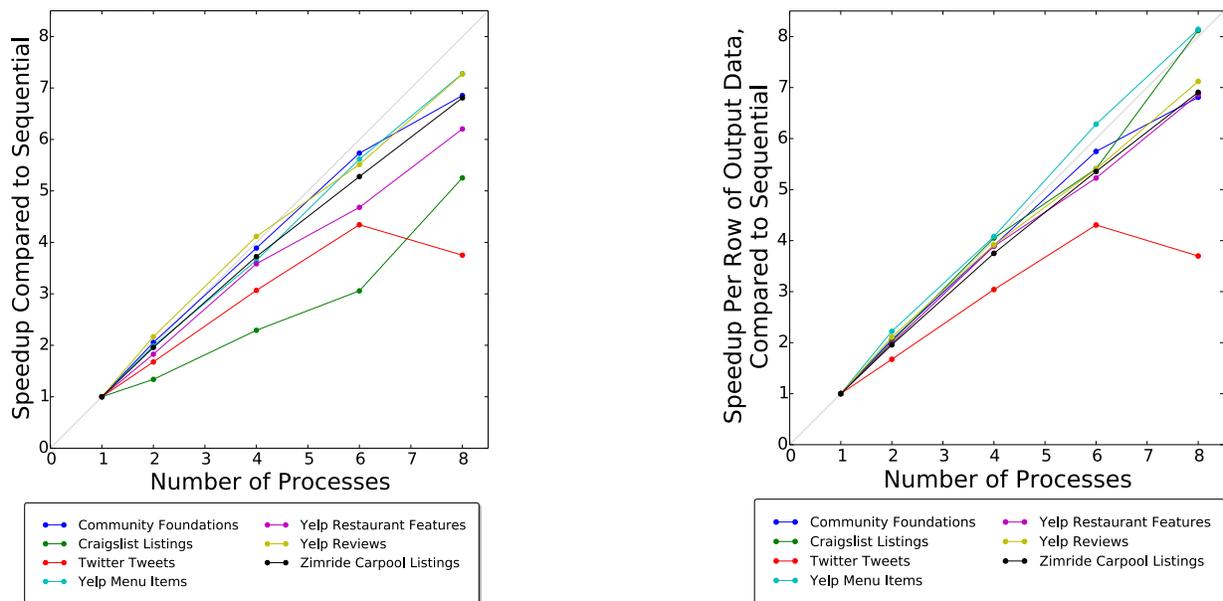
Figure 6.1: **Hash-based** signature parallelization performance. Each line represents one of our benchmarks. The light gray diagonal represents the ideal speedup. Effective speed, our preferred measure of execution speed, reveals we achieve irregular speedups. Performance is worse for benchmarks with subtasks of varying sizes.

the amount of work, we also chart the *effective speed*: the execution time per row of scraped output data. This is the critical metric, since it reflects how quickly an execution does its allotted work. Again, we normalize by the effective speed of the serial variant. The resultant speedups appear in Figure 6.1b.

Once we adjust for the amount of work completed by each run, the ‘Craigslist Listings’ performance comes quite close to the ideal. Craigslist tends to uphold the same-size assumption, so its poor performance is attributable to the greater amounts of data that multi-worker versions can access. However, it is clear from the other benchmarks that the amount of work being completed is not the driving explanatory force for the execution times. Rather, the final execution time reflects the execution time of the last-running, longest-running process—whichever process was assigned longer subtasks.

Lock-Based Runtime

Hypothesis. We expect that our lock-based approach should scale well for tasks with large numbers of subtasks.



(a) **Time (Parallel Execution).** Run time for benchmark executions, normalized by serial run times.

(b) **Effective Speed (Parallel Execution).** Run time per row of output data, normalized.

Figure 6.2: **Lock-based** signature parallelization performance. Each line represents one of our benchmarks. The light gray diagonal represents the ideal speedup. Effective speed, our preferred measure of execution speed, reveals that even parallelizing on a single machine achieves near-perfect speedup for tasks that are not CPU-bound (Subfigure b).

Procedure. We evaluated the lock-based signature technique by running each benchmark script with 1, 2, 4, 6, and 8 workers. Figure 6.2a shows the execution times for all benchmark-workers pairs, normalized by the execution time for the sequential variation, using outermost locking.

Results. Figure 6.2a reveals that for five of the seven benchmarks, speedups approach the ideal speedups. Exceptions are the ‘Twitter Tweets’ and ‘Craigslist Listings’ benchmarks. For the Twitter case, the explanation is simple: the Twitter benchmark quickly becomes CPU-bound because of how large (in terms of DOM nodes, not in terms of screen inches) a Twitter tweetstream webpage becomes as one scrolls to the 1,000th tweet. (The evaluation machine had 8 hyperthreads, so we did not expect CPU-intensive workloads to scale well.) The Twitter performance suggests the potential to optimize the Helena language for large DOM trees; however, it also motivates our desire to offer distributed execution for UE scrapers since distribution across machines can accelerate cases like Twitter that exhibit unusual patterns and may not be worth the trouble of pattern-specific optimizations.

Again, our preferred measure is effective speed, which adjusts for the amount of work completed by each parallel run. Effective speedups appear in Figure 6.2b. Twitter remains an outlier since its

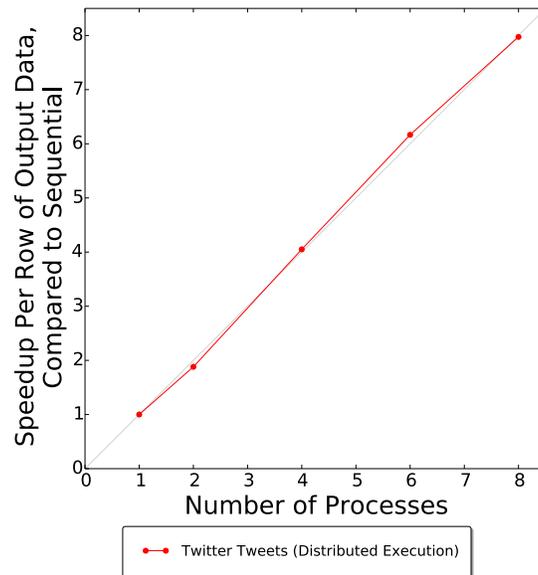


Figure 6.3: **Effective Speed (Distributed Execution)**. Run time per row of output data, normalized. With distributed execution, we can achieve near-perfect speedup even for tasks that are CPU-bound.

poor performance was caused not by additional work but by CPU-boundedness. However, with the exception of Twitter, we see that the speedups cluster quite close to the diagonal that represents ideal speedup. Since we expected that most tasks were dominated by network latency rather than CPU, these benchmarks largely exhibit the pattern we anticipated. Overall lock-based speedups—once we account for the increase in the amount of work that multi-worker variations complete—were indeed very close to the ideal speedups.

Distributed Execution

Hypothesis. We expect that scripts that do not achieve near-ideal speedups in a parallel context may still achieve near-ideal speedups in a distributed context.

Procedure We used the outermost lock-based runtime implementation to run distributed executions of the ‘Twitter Tweets’ benchmark with 1, 2, 4, 6, and 8 machines, each running a single process.

Results Although we anticipated that most scripts would not be CPU-bound, we were surprised to find that only one of our long-running benchmark scripts failed to produce near-perfect speedups in the parallel context. Since only the Twitter benchmark exhibited this behavior, we can provide only limited data on this question. Figure 6.3 shows the effective speedup for the ‘Twitter Task’ in the distributed setting. Comparing its performance in the distributed context (Figure 6.3) with its performance in the parallel context (Figure 6.2b), it is clear that distributing the execution can

indeed allow us to produce near-ideal speedups even for tasks that perform poorly in parallel settings. This is only one task, so we cannot generalize well from this data. However, we hypothesize that in cases like the Twitter case, where CPU-boundedness is the reason for the poor parallel performance, distributing the execution is likely to produce good performance.

We conducted one additional distributed experiment with *small-scale* web scraping tasks to evaluate whether our approach, designed for long-running tasks, also improves performance for short-running tasks. We ran each scraper with one worker and with 100 workers using outermost locking:

Benchmark	Sequential Time	Distributed Time	Speedup
Song Lyrics	25 minutes and 18 seconds	30 seconds	50
Stackoverflow Questions	46 minutes and 49 seconds	38 seconds	74
Newspaper Event Calendar	47 minutes and 3 seconds	37 seconds	76
Recipes	50 minutes and 45 seconds	42 seconds	72

With such short-running tasks (with execution times in the seconds range), start-up time becomes a major determinant of the run time and begins to strongly affect speedups. Still, even without optimizing our implementation for fast start-up performance, we see speedups in the $50\times$ to $76\times$ range.

Case Study: A Comparison of Approaches

For a sense of how our signature-based approach fits into the broader space of web scraping parallelization techniques, we hand-coded some task-specific techniques for a case study. Our case study of choice is a task that scrapes information about all votes cast in Peru’s 2016 presidential election from the website of Peru’s National Office of Electoral Processes [70]. The results are broken down by state, province, district, polling place, and ballot box. To access results for a given box, a website user selects the correct department, province, district, and polling place from pulldown menus. The user waits between each selection for the webpage to communicate with the server and update the page, then waits for the page to load the relevant list of ballot boxes, then clicks on the target box. This click updates the page with the voting results for the box. The top-level URL of the webpage is the same throughout. The dataset includes tens of thousands of ballot boxes, and scraping it sequentially takes over a week.

We selected this case study for a number of reasons. First, the fact that this dataset takes so long to collect makes it exactly the kind of task users most want to accelerate. Second, the political scientist who requested this dataset is technically sophisticated; he had written Python scripts to scrape large amounts of information about Peruvian laws from another website. He attempted to write a script for this election data himself, but he found that the amount of AJAX updating involved was an insurmountable obstacle. This suggests the task involves the kind of reverse engineering that even technically sophisticated social scientists and data scientists are unwilling to do.

Hypothesis 1. Because repeating UI interactions is expensive, we expect an approach that parallelizes at the level of coarse-grained tasks will be faster than an approach that parallelizes at the level of the finest-grained tasks.

Hypothesis 2. Because a signature-based UE script must complete all AJAX requests plus the additional work of UI interaction, we expect that a UE implementation will be slower than a hand-written AJAX parallelization approach.

Procedure. We compared our lock-based approach against several designs for hand-written alternatives. In particular, we compare against two scraping approaches that do not use UI manipulation: URL-based parallelization and AJAX response-based parallelization. We also compared against a variant that uses UI interactions, as our signature-based program does, but parallelizes at the finest-granularity tasks rather than the coarsest-granularity tasks. We use this comparison to understand how much benefit we gain from using a technique that avoids reproducing common prefixes of UI interactions more than once per worker process. (With the outermost locking technique, a worker iterates through each item of the outermost loop exactly once, executing or not executing each item. In contrast, a process that uses UI interactions to restore state and parallelizes at the finest-granularity level must iterate through the 100th item of the outermost list every time it aims to reach a subtree of items at the 100th index or beyond. We aim to evaluate the cost of repeating these shared task prefixes.)

For this case study, the URL remains the same throughout all interactions with the webpage. Thus, a URL-based parallelization approach cannot be used for this task. Our comparison with this strategy ends at the observation that this strategy is not applicable to this target dataset.

Both the AJAX and fine-grained UI approaches are applicable. We hand-coded programs that implement *only the final parallel phase* of these approaches. Both versions have a queue that stores all the information they need to reach information for a given ballot box. For the AJAX version, the queue contains the parameters for the necessary AJAX requests. For the UI approach, it contains the XPaths of pulldown menus and the values to which they should be set, and also the text associated with the target ballot box's link (for reaching the details of the votes cast in a given ballot box). For the purposes of this limit study, we collected this queue information from a run of our signature-based program and fed it into the hand-coded programs as input to produce the queue. In practice, the programs would also need to collect this queue information, which means the programs would in fact run longer.

The aim is to do a limit study bounding how much benefit we can gain from these approaches, rather than comparing the full execution times. We made this choice in large part because writing the programs for doing the full versions for a hierarchy this deep is very time consuming and tedious. Even with the workaround that let us test only the final parallel phase, developing the AJAX version took 4 hours and 21 minutes of active development time and the UI version took 2 hours and 29 minutes of active development time. The process of debugging their mid-script errors stretched over days. Comparing partial runs of these alternative strategies against full runs of signature-based scraping handicaps the signature-based approach, but for the purposes of a limit study, this is sufficient.

We further handicapped the signatures approach by using the non-adaptive outermost lock-based approach and limiting the target dataset to data from only the first 8 elements of the outer loop and running all scraping executions with 8 workers. This is still a very large dataset, with voting

Approach	Applicable?	Execution Time
URL Parallelization	No	-
AJAX Parallelization (Last Phase Only)	Yes	10.5 hours
UI Manipulation: Fine-Grained (Last Phase Only)	Yes	24+ hours*
UI Manipulation: Signature-Based Coarse-Grained	Yes	14.9 hours

*Timed out at 24 hours.

Table 6.3: **Comparison of four techniques** for parallelizing our Peruvian government case study task. ‘Applicable?’ indicates whether the strategy can be applied to the task. ‘Execution Time’ indicates how long the resultant program took to collect the target data. By running only the final parallel phases of the AJAX and UI strategies, we get a lower bound on how long those programs would run. In contrast, the signature-based variant runs the entire task.

results from more than 19,000 ballot boxes. But the limitation to the first 8 elements means that our simple lock-based implementation was limited in how well it could load balance—and indeed, the 8 workers did ultimately complete very different amounts of work.

Results Table 6.3 shows how long the variants ran.

This task should be especially amenable to the fine-grained UI-based strategy since most of the navigation required to reach a subtask occurs on a single page; in contrast to the many tasks where a UI-based navigation approach requires iterating through thousands of items and clicking on a ‘Next’ button hundreds of times, this task should be unusually amenable to a UI task queue strategy. Nevertheless, the UI-based variation timed out. The amount of redundant UI interaction necessary for this approach made even the final parallel phase slower than the whole of our coarse-granularity approach. This behavior was the motivation for our queue-free design in which workers determine whether to execute or pass over tasks upon encountering them, so this is the behavior we expected.

While our signature-based approach was faster than the fine-grained UI strategy, it was slower than the final phase of AJAX-based parallelization. Again, this is the expected behavior. All AJAX requests made in the AJAX script must also be made by the signature-based script—albeit triggered by UI interactions—and our truncated version of the dataset prevented the signature-based program from doing good load balancing, while the AJAX strategy was allowed to load balance at the ballot box level. In short, it would not be reasonable to expect our approach to outperform the hand-coded AJAX approach. However, the difference in execution times is small enough that we expect some programmers will prefer to run the longer-running signature approach in order to avoid spending the programming time required to reverse engineer a site’s AJAX communication. Recall that with the signature approach, the programmer reasons about what makes two districts the same—e.g., they share a name—while the AJAX approach requires them to identify AJAX requests that load a district’s details, extract IDs for each district over which they will iterate from webpage internals, then parse the AJAX response to extract inputs for the next AJAX request. The choice between the two approaches is likely to depend on the expertise of the programmer, the difficulty of reverse engineering a particular target webpage, and how much the programmer values performance.

Since the focus of this work is on UE scrapers, we are most interested in the comparison of signature-based approaches with the fine-grained locking approach. This result confirms the value

of using a signature-based approach that executes or passes over each node immediately upon finding it, the value of avoiding repetition of shared UI interaction prefixes.

6.5 Related Work

Web Automation Parallelization

Web scraping languages typically do not offer built-in parallelization strategies. Most scraping DSLs are implemented as libraries for general-purpose programming languages: Selenium [83] for C#, Groovy, Java, Perl, PHP, Python, Ruby, and Scala; BeautifulSoup [73] and Scrapy [80] for Python; Nokogiri [63] and Hpricot [28] for Ruby; HXT [26] for Haskell; and so on. Users of these tools can pick from among whatever language support the host language provides for parallelism.

We know of a few language constructs explicitly designed to parallelize tasks related to web automation. In particular, RCurl's `getURIAsynchronous` takes as input multiple URLs to download; it submits all requests, then processes the replies as data becomes available on each connection, interleaving response processing [45]. The RCrawler library is primarily focused on crawling, narrowly defined, rather than scraping or automation more broadly. However, it offers very clean parallelization for a particular crawling workflow in which each task is to GET one URL, extract links, canonicalize links, then add them to the work queue [36]. Although its programming model is quite restrictive, the RCrawler library clearly qualifies as a DSL for parallel web interactions. These constructs can be used to parallelize URL-based scrapers but not user-emulating scrapers; they rely on URLs for migrating tasks and thus do not handle transferless parallelism.

Irregular Parallelism

Transferless parallelism is loosely related to irregular parallelism [41, 68, 27, 40] in that the input data is a pointer-based data structure, a tree. However, prior approaches for irregular parallelism depend on the following assumptions:

1. The graph can be updated, *but only by the parallel algorithm*, which is controlled by the programmer. In our domain, the graph is changed during execution by the environment (the web server). In standard irregular programs, the parallel algorithm can use locks to prevent any agent from altering data; we cannot. Likewise, standard irregular programs can trivially mark visited nodes; we cannot.
2. All workers share a single graph representation. In our domain, in the limit, the web server can deliver different trees to all workers.
3. Workers can pass around pointers to nodes. In our domain, we have no pointers to nodes.

It is the breaking of these assumptions that drives our approach of having each worker: (1) claim tasks via custom unique IDs, (2) walk over the graph using the same traversal strategy, and (3) process a node upon first visiting.

Work Sharing and Work Stealing

For transferless parallelism, we cannot use generic work sharing or work stealing [25, 11, 12] because we cannot transfer a task encountered by worker A to worker B. The lack of an efficient mechanism for transferring a task—for restoring state—is the primary obstacle that drives our design. There do exist inefficient mechanisms for transferring state. In particular, we can repeat a prefix of UI manipulations to restore state. For our election data case study (Section 6.4), we made one variation that used work sharing with a queue of tasks that restored state via UI interactions; this is the only general-purpose approach for restoring state short of cloning the browser. The program timed out, taking at least 9 hours longer than the signature-based program that processed tasks as it encountered them. In some cases, even this strategy is even impossible, if the web server displays a given node to one worker but not to others. Because we lack efficient task pointers, it is inefficient to use any form of work queue in this domain. This motivates the use of approaches in which workers decide whether to claim or pass over nodes during traversals of the data.

6.6 Conclusion

For user-emulating web scrapers facing the constraints of transferless parallelism, the signature-based technique offers a template for performant parallelized scrapers. With near-ideal speedups in single-machine contexts and speedups in the $50\text{--}76\times$ range in distributed contexts, we conclude that signature-based parallelization is practical for real scraping tasks and that it is a good fit for the unusual challenges associated with non-transferable work and inconsistent workloads.

Chapter 7

Skip Block Usability Evaluation

In this chapter, we evaluate whether programmers and non-programmers can learn and use skip blocks effectively.

Writing Skip Blocks

Q1: Given just the information that a user will have upon first demonstrating a script, can users identify what set of features uniquely identifies an entity?

H1: We expect users will do well overall, but miss some corner cases.

Procedure

For each benchmark in our benchmark suite, we collected the attributes of the first object observed by each skip block construct. For instance, for the Yelp Menu Items benchmark, this is the first restaurant (for the restaurant block) and the first menu item of the first restaurant (for the menu item block). This is the same information that a user is guaranteed to have during script demonstration, and is in fact the same information that we show in the tool’s current UI for adding new skip blocks. We constructed a survey in which users were asked to select the set of columns (the object attributes) that can be used to uniquely identify a given object based on this first row of data. For most tasks, they were also given links to the source webpages which they could use to observe more instances of the entity. See Figure [7.1](#) for a side-by-side comparison of the real tool UI and the survey question. We made the survey available as an online survey and recruited participants through social networks and mailing lists. Participants were not compensated for their time. We recruited 35 participants (16 non-programmers, 19 programmers).

Before releasing the survey, we selected ground truths. For each entity, we chose *two* ground truths—two sets of columns that could serve as the ‘correct’ unique identifier. Correctness in this context is somewhat fuzzy in that one user may want to rescrape a tweet each time it has a new number of retweets, while another user may only be interested in the text, which does not change over time. Since giving participants a single sharp, unambiguous notion of entry equality for each task essentially boils down to revealing the target solution, we had to allow them to supply their

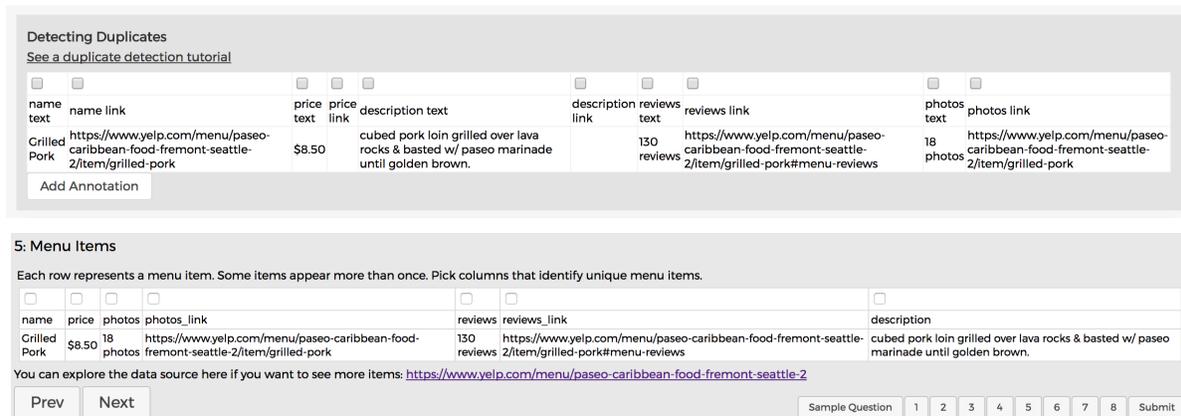


Figure 7.1: The UI for adding a new skip block in the real tool (top) and the same task in the survey for our user study (bottom).

own definitions of duplicates. Directions could not be more specific than ‘pick columns that identify unique <entities >.’

Given the inherent fuzziness, there is no one correct answer, or even two correct answers. Acknowledging this, we elected to use two ground truths that we felt would cover two important interpretations of the duplicate idea. The first set of ground truths, the Site-Based Conservative Standard, attempts to use the websites’ own notions of uniqueness to identify duplicates; if a site provided a unique id, or a URL that includes a unique id, this ground truth relies on those attributes, often ignoring similarities that might cause a human observer to classify two rows as duplicates. The second set of ground truths, the Human-Based Aggressive Standard uses the authors’ own perceptions of what should qualify as a duplicate, based on having seen the output data. This approach is more aggressive than the Site-Based Standard, collapsing more rows. With the Human-Based Standard, two Craigslist postings with different URLs but the same apartment size, price, and advertisement text are collapsed, and two Yelp reviews with different URLs but the same star rating and review text are collapsed. Importantly, we selected the ground truths before soliciting any survey responses.

For each user-suggested skip block, we identified the rows that the ground truths would remove but the user suggestion leaves, and also the rows that the ground truths would leave but the user suggestion removes.

Results

On average, participants spent 0.93 minutes on each annotation task and 4.0 minutes on the instructions. See Figure 7.2. Programmers learn to use skip blocks in 1.9 minutes on average, after which they write each new skip block in 51 seconds on average. For non-programmers, the learning process averages 6.5 minutes and each new skip block averages 61 seconds.

We are primarily interested in two measures: i) how many rows each user annotation keeps but the Conservative standard discards as duplicates, and ii) how many rows each user annotation

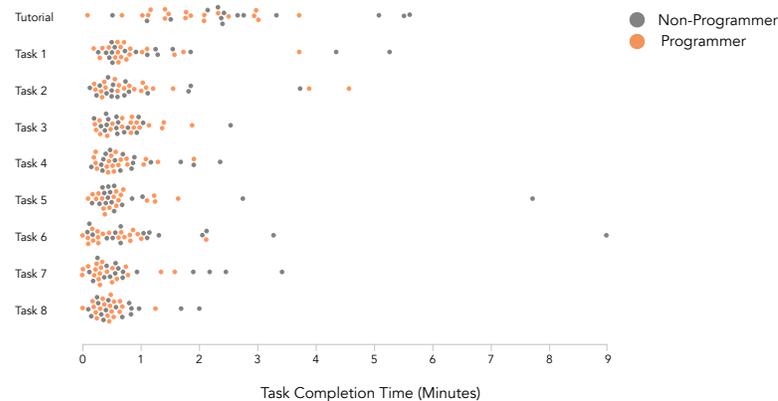


Figure 7.2: **Time to Learn About Skip Blocks and Time to Add A New Skip Block.** Orange dots represent programmers, and gray dots represent non-programmers. Non-programmers add a new skip block—parallelize a new program—in 61 seconds on average, non-programmers in 51 seconds on average.

discards but the Aggressive standard keeps. We call these keeping and removal errors. For all entities, all participants produced 0 keeping errors relative to the Conservative standard; the average removal error rate ranges from 0.00% (Community Foundations, Twitter Accounts) to 5.8% (Zimride), with a median of 1.3%. See Figure [7.3](#) in the next subsection for a visualization of error rates by benchmark.

For a deeper understanding of what these error rates represent, we discuss the most common causes of errors. Since users produce no keeping errors relative to the Conservative standard, we focus on removal errors. Removal errors were most common for Zimride carpool listings and Yelp menu items.

The most serious Zimride error occurred when users collapsed all listings posted by a given username. Essentially, they made the assumption that each user would only post one carpool listing, while in fact a given user may request or offer many different rides—to multiple different places or at multiple different times of day. Surprisingly, even requiring all of [‘carpool_endpoints’, ‘user_name’, ‘carpool_times’, ‘carpool_days’] to match produces some removal errors; this stems from the fact that a few users post the same commute both as a passenger and as a driver. It is worth noting that for the Zimride case, users were at a disadvantage. Because the carpool data came from an institution-specific Zimride site, it was behind a login, and thus this was one of the cases for which we provided no link to the source data. For this reason, participants could not explore additional rows of data within the site.

The most serious Yelp menu errors occurred when participants relied on different menu items having different descriptions, photo links, or review links. On the face of it, this seems reasonable, but in practice many menu items lack all three of these attributes, so all items that lack the attributes are collapsed.

These errors suggest directions for future debugging tools. For instance, one simple debugging

tool could show users two objects for which all key attributes match but all other attributes diverge, then ask “did you really mean to treat these objects as duplicates?” This would allow users to quickly identify issues with answers like the Yelp menu photo link mistake. Seeing two rows with different food item names, different prices, and different descriptions collapsed because they lack a photo link would quickly remind users that missing data can subvert their answers. A debugging tool like this is easy to build and easy to use [57].

Overall, we are pleased to observe that participants can quickly produce effective skip blocks using the same UI that we offer in the current tool, without additional tool support. We are also encouraged that removal errors are more common than keeping errors, since these are the errors that future tool support can most easily address.

Programmers vs. Non-Programmers

Q2: Do programmers perform better on the attribute selection task than non-programmers?

H2: We expect programmers will perform slightly better than non-programmers on this task.

We expect that programmers will have had more reasons to work with unique identifiers and also that they may have more exposure to web internals and thus be likelier to recognize sites’ own internal unique identifiers in links. With this in mind, we hypothesized that programmers would slightly outperform non-programmers.

Procedure

Our survey asked participants to indicate whether they are programmers or non-programmers.

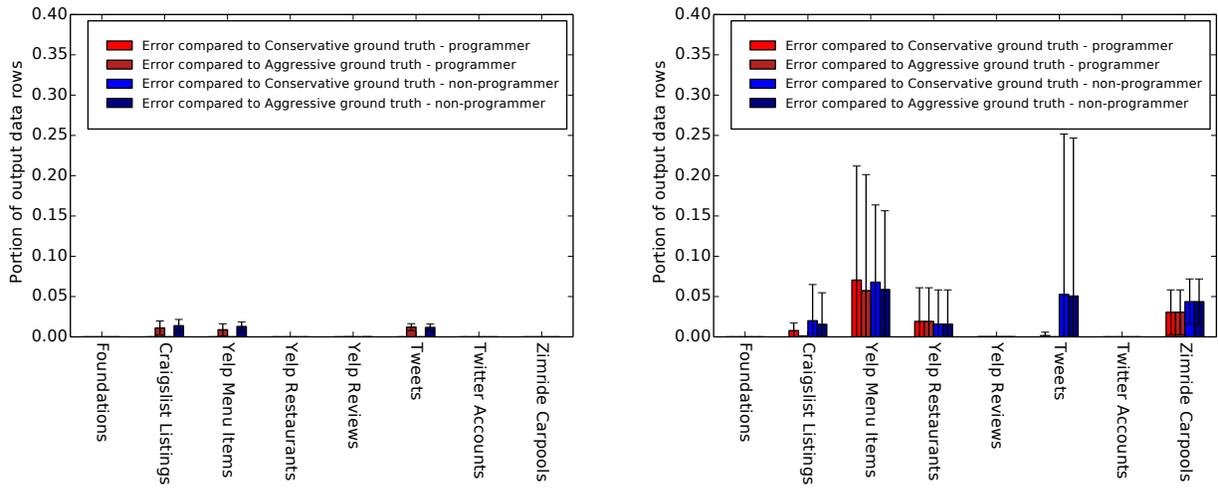
Results

Our hypothesis that programmers will perform better than non-programmers is not supported.

Figure 7.3 shows a comparison of programmer and non-programmer error rates. Overall, we see little difference between programmers and non-programmers, except in the case of the Tweets entity. As it turns out, the high non-programmer bars for the Tweets entity reflect the results of 15 non-programmers with removal error rates under 0.001 and a single non-programmer with an error rate of 81%—this participant elected to use only the display name and username attributes, apparently interpreting the goal as identifying unique Twitter accounts, rather than unique tweets. (It may be worth noting that this particular user took less than 4 and a half minutes to complete the entire survey, including reading the instructions page and completing all eight annotation tasks.)

The average programmer had a keeping error rate of 0% and a removal error rate of 1.365%. The average non-programmer had a keeping error rate of 0% and a removal error rate of 2.322%. The difference between the two groups is not statistically significant ($p=.32$ for removal errors).

Programmers and non-programmers also spent similar amounts of time on each annotation task. Programmers spent 0.86 minutes on average, while non-programmers spent 1.02 minutes on average.



(a) **Keeping errors** for programmers and non-programmers. When a user answer keeps a row that the ground truth does not keep, this is one keeping error. We see that users do not make keeping errors with respect to the Conservative standard—anything the site recognizes as a duplicate, users also recognize. Relative to the Aggressive standard, users do make errors; this suggests that, as expected, they choose a point between the Conservative and Aggressive standards.

(b) **Removal errors** for programmers and non-programmers. When a user answer removes a row that the ground truth does not remove, this is one removal error. We see that removal errors are more common than keeping errors. Unsurprisingly, users are more aggressive than the Site-based Conservative standard, but it appears they are also slightly more aggressive than our Aggressive Standard.

Figure 7.3: A comparison of programmers with non-programmers, based on whether they are more or less aggressive than the Conservative and Aggressive standards.

Given that non-programmers can produce an average error rate of less than 3% in the context of a 12 minute online survey (and despite lacking familiarity with the data to be scraped), we are satisfied that end users will be able to use our construct for their scraping tasks.

Chapter 8

Conclusion: Programming Tools for the Future of Data Science

From the perspective of my social science collaborators, the key contribution of Helena is removing their dependence on programmers. In building Helena, we took a per-domain approach instead of a per-project approach (Figure 8.1). Rather than hearing a team’s particular goal, then building one scraper for the team, we built a tool that puts a whole class of programs in easy reach for a broad swath of non-technical domain experts. With Helena in hand, the teams of domain experts no longer need to hire or rely on a programmer for web data collection.

Going forward, I hope that we can make this same transition in many more programming domains. We can move away from the situation where a domain expert needs to find an amenable coder, explain the relevant domain expertise, then wait for the coder to produce a program. We can move towards the situation where domain experts produce their target programs themselves. Cases where domain experts are already collaborating with programmers on a per-project basis can serve as a guide, pointing to the domains in which we need new programming tools to support the future of data science.

Our Helena experience suggests strategies for developing the next generation of programming tools for non-coders and novice coders. From the perspective of PBD, the key contribution of Helena is actually that we departed from pure PBD. Helena succeeded because it is *not* strict PBD. It combines demonstration-based authoring with direct program edits, giving users multiple ways to communicate their intent. However, this DORA workflow is only one point in a vast design space of tools for *Programming By X (PBX)*—programming by anything and everything the user can communicate to a computer. End users are much less restricted than we have always assumed in what inputs they can provide, and there are many promising specifications at hand: coached demonstrations, formalized domain expertise, direct question answering, and fuzzy outputs to name a few. PBX tools should accept all of these as specifications.

Many web automation problems remain open. Helena produces likely relations by layering a Relation Selector on top of a traditional Relation Finder because (i) input demonstrations do not directly produce labeled table data and (ii) traditional Relation Finders perform poorly with only a single row of sample data. However, we may be able to do better by abandoning traditional Relation

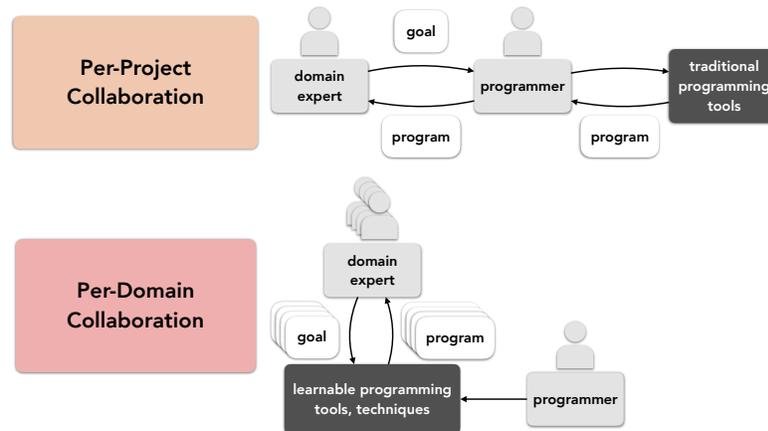


Figure 8.1: Per-Project versus Per-Domain Collaboration Styles. A per-project collaboration style delivers the programs that end users need, but it keeps the non-technical domain experts dependent on programmers. A per-domain collaboration style—such as the Helena collaboration, centered on the web automation domain—empowers many users with different needs to write their own programs.

Finders altogether and starting from scratch. There is also much more to learn about how users can use DORA tools to produce large programs—presumably not with a single demonstration. Can demonstrations be combined the way we compose library functions? And we have barely scratched the surface when it comes to studying the Helena language’s editability. How should synthesizers identify maximally editable programs, how much of Helena’s benefit comes from the synthesizer versus program editability, and how do non-traditional programmers respond to various editing tasks? There are also many exciting potential tools that come in range now that we have robust web PBD. We have used Helena to build a do-it-yourself customizable voice assistant; users give Helena the prompt they plan to speak (“Is my pool open?” “What are the seasonal cupcakes this month?”), then demonstrate how to find the information Helena should use to answer. We have hooked Helena scripts up to web interfaces, IoT devices, and Arduinos. Since so many of our daily activities are now web-mediated, we can use web automation for actuation—to alter server state, control a device, or direct a digital agent. The possibilities for automatic webpage testing tools, now that we have an easy way of generating robust web automation programs, are endless. What other tools come into range, now that we have this new enabling technology?

Even more importantly, many Usable Programming (UP) problems remain open. There is a tremendous gap between the programming skills of occasional programmers—social scientists, journalists, data scientists—and the skills required to write the programs they want. But the need is pressing. Programming is changing. While there are about 20 million programmers in the world, there are now at least twice as many individuals performing end-user programming [78]. Coders with traditional programming training are already only a small sliver of the people out in the world writing programs. This reality is starting to change how we teach, via data science curricula. It should also change how we design and build programming languages, programming environments,

and programming tools.

My discussions with social and data scientists have revealed many outstanding programming needs. These users represent only a small slice of the population of end users, but even in this one slice, the programming needs are diverse and complex. To name a few, they need: tools for designing and implementing communication-based experiment workflows and online experiment workflows; chatbot design tools; easy access to computer vision algorithms; easy access to natural language processing; data storage and schema matching solutions adapted to their needs and expertise.

Most, maybe even all, of these target domains are abstraction-resistant. We can imagine high-level DSLs describing what we want programs to do, but it is not clear how to compile down to the low-level code we would need to run them. As in the web automation domain, users can reason about the high-level logical structure of programs in these domains but cannot use traditional programming tools to provide all requisite implementation details. PBX may be a good fit for these domains. The target audience can provide examples, demonstrations, coached demonstrations, domain expertise. They can describe what they want but not how to get there. In short, they could use PBX tools.

Even in the domain of web automation, many important Usable Programming questions remain. However, Helena demonstrates that Programming By X is now a practical approach for writing real programs, putting large and complex programming tasks in reach for non-programmers. It is already clear that, together, PL and HCI hold the potential to blur the line between programmers and non-programmers.

Bibliography

- [1] *A guide to preventing Webscraping*. <https://github.com/JonasCz/How-To-Prevent-Scraping>. July 2017.
- [2] a9t9. *Web Browser Automation with KantuX, Desktop Automation, OCR - Fresh 2017 Robotic Process Automation (RPA)*. Mar. 2018. URL: <https://a9t9.com/> (visited on 03/28/2018).
- [3] Brad Adelberg. “NoDoSE - A tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents”. In: *SIGMOD Record*. 1998.
- [4] Naveen Ashish and Craig A. Knoblock. “Semi-Automatic Wrapper Generation for Internet Information Sources”. In: *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*. COOPIS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 160–169. ISBN: 0-8186-7946-8. URL: <http://dl.acm.org/citation.cfm?id=646744.703555>.
- [5] *AskF5 | Manual Chapter: Detecting and Preventing Web Scraping*. https://support.f5.com/kb/en-us/products/big-ip_asm/manuals/product/asm-implementations-11-4-0/22.html. Aug. 2018.
- [6] Nikolaos Askitas and Klaus F. Zimmermann. “The internet as a data source for advancement in social sciences”. In: *International Journal of Manpower* 36.1 (2015), pp. 2–12. DOI: [10.1108/IJM-02-2015-0029](https://doi.org/10.1108/IJM-02-2015-0029), eprint: <https://doi.org/10.1108/IJM-02-2015-0029>. URL: <https://doi.org/10.1108/IJM-02-2015-0029>.
- [7] Jean-Eric Aubert. “Into the future with social sciences”. In: *OECD Observer* 217/218 (June 1999), p. 78.
- [8] Shaon Barman. “End-User Record and Replay for the Web”. PhD thesis. University of California, Berkeley, 2015.
- [9] Shaon Barman et al. “Ringer: Web Automation by Demonstration”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 748–764. ISBN: 978-1-4503-4444-9. DOI: [10.1145/2983990.2984020](https://doi.org/10.1145/2983990.2984020). URL: <http://doi.acm.org/10.1145/2983990.2984020>.
- [10] D. J. Bernstein. *D. J. Bernstein*. <http://cr.yp.to/djb.html>. Nov. 2017.

- [11] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: [10.1145/324133.324234](https://doi.org/10.1145/324133.324234). URL: <http://doi.acm.org/10.1145/324133.324234>.
- [12] Robert D. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: [10.1145/209937.209958](https://doi.org/10.1145/209937.209958). URL: <http://doi.acm.org/10.1145/209937.209958>.
- [13] Chia-Hui Chang et al. “A Survey of Web Information Extraction Systems”. In: *IEEE Trans. on Knowl. and Data Eng.* 18.10 (Oct. 2006), pp. 1411–1428. ISSN: 1041-4347. DOI: [10.1109/TKDE.2006.152](https://doi.org/10.1109/TKDE.2006.152). URL: <http://dx.doi.org/10.1109/TKDE.2006.152>.
- [14] Kerry Shih-Ping Chang and Brad A. Myers. “Gneiss”. In: *J. Vis. Lang. Comput.* 39.C (Apr. 2017), pp. 41–50. ISSN: 1045-926X. DOI: [10.1016/j.jvlc.2016.07.004](https://doi.org/10.1016/j.jvlc.2016.07.004). URL: <https://doi.org/10.1016/j.jvlc.2016.07.004>.
- [15] Sarah Chasins, Maria Mueller, and Rastislav Bodik. “Rousillon: Scraping Distributed Hierarchical Web Data”. In: *Proceedings of the 31st annual ACM symposium on User interface software and technology*. UIST ’18. Berlin, Germany: ACM, 2018. DOI: [10.1145/3242587.3242661](https://doi.org/10.1145/3242587.3242661). URL: <https://doi.org/10.1145/3242587.3242661>.
- [16] Sarah Chasins et al. “Browser Record and Replay As a Building Block for End-User Web Automation Tools”. In: *Proceedings of the 24th International Conference on World Wide Web Companion*. WWW ’15 Companion. Florence, Italy, 2015, pp. 179–182. ISBN: 978-1-4503-3473-0. DOI: [10.1145/2740908.2742849](https://doi.org/10.1145/2740908.2742849). URL: <http://dx.doi.org/10.1145/2740908.2742849>.
- [17] Allen Cypher. “EAGER: Programming Repetitive Tasks by Example”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’91. New Orleans, Louisiana, USA: ACM, 1991, pp. 33–39. ISBN: 0-89791-383-3. DOI: [10.1145/108844.108850](https://doi.org/10.1145/108844.108850). URL: <http://doi.acm.org/10.1145/108844.108850>.
- [18] M. Fischer, S. Lyon, and D. Zeitlyn. “The Internet and the future of social science research.” In: *The Sage handbook of online research methods*. Ed. by N. Fielding, R.M. Lee, and G. Blank. London: Sage, June 2008, pp. 519–536. URL: <http://dro.dur.ac.uk/5034/>.
- [19] Sergio Flesca et al. “Web Wrapper Induction: A Brief Survey”. In: *AI Commun.* 17.2 (Apr. 2004), pp. 57–61. ISSN: 0921-7126. URL: <http://dl.acm.org/citation.cfm?id=1218702.1218707>.
- [20] Tim Furche et al. “Robust and Noise Resistant Wrapper Induction”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 773–784. ISBN: 978-1-4503-3531-7. DOI: [10.1145/2882903.2915214](https://doi.org/10.1145/2882903.2915214). URL: <http://doi.acm.org/10.1145/2882903.2915214>.
- [21] Matthew Gentzkow and Jesse M. Shapiro. “Code and Data for the Social Sciences: A Practitioner’s Guide”. In: 2014.
- [22] Greasemonkey. *Greasemonkey :: Add-ons for Firefox*. <https://addons.mozilla.org/en-us/firefox/addon/greasemonkey/>. Nov. 2015.

- [23] Michael Grishaber. *The Rise of the Single Page Application*. June 2016. URL: <https://merlinone.com/7006-2/>.
- [24] Sumit Gulwani. “Automating String Processing in Spreadsheets using Input-Output Examples”. In: Jan. 2011.
- [25] Robert H. Halstead Jr. “Implementation of Multilisp: Lisp on a Multiprocessor”. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 9–17. ISBN: 0-89791-142-3. DOI: [10.1145/800055.802017](https://doi.org/10.1145/800055.802017). URL: <http://doi.acm.org/10.1145/800055.802017>.
- [26] HaskellWiki. *HXT - HaskellWiki*. https://wiki.haskell.org/HXT#Selecting_text_from_an_HTML_document. Nov. 2017.
- [27] Nikhil Hegde, Jianqiao Liu, and Milind Kulkarni. “SPIRIT: A Runtime System for Distributed Irregular Tree Applications”. In: *SIGPLAN Not.* 51.8 (Feb. 2016), 51:1–51:2. ISSN: 0362-1340. DOI: [10.1145/3016078.2851177](https://doi.org/10.1145/3016078.2851177). URL: <http://doi.acm.org/10.1145/3016078.2851177>.
- [28] hpricot. *hpricot/hpricot*. <https://github.com/hpricot/hpricot>. Aug. 2015.
- [29] Chun-Nan Hsu and Ming-Tzung Dung. “Generating finite-state transducers for semi-structured data extraction from the Web”. In: *Information Systems* 23.8 (1998). Semistructured Data, pp. 521–538. ISSN: 0306-4379. DOI: [http://dx.doi.org/10.1016/S0306-4379\(98\)00027-1](http://dx.doi.org/10.1016/S0306-4379(98)00027-1). URL: <http://www.sciencedirect.com/science/article/pii/S0306437998000271%22%7D>.
- [30] Darris Hupp and Robert C. Miller. “Smart bookmarks: automatic retroactive macro recording on the web”. In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*. UIST '07. Newport, Rhode Island, USA: ACM, 2007, pp. 81–90. DOI: [10.1145/1294211.1294226](https://doi.org/10.1145/1294211.1294226). URL: <http://doi.acm.org/10.1145/1294211.1294226>.
- [31] David F. Huynh, Robert C. Miller, and David R. Karger. “Enabling web browsers to augment web sites’ filtering and sorting functionalities”. In: *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM Press, 2006, pp. 125–134. ISBN: 1595933131. DOI: [10.1145/1166253.1166274](https://doi.org/10.1145/1166253.1166274). URL: <http://dx.doi.org/10.1145/1166253.1166274>.
- [32] Import.io. *Import.io | Web Data Platform & Free Web Scraping Tool*. Mar. 2016. URL: <https://www.import.io/> (visited on 03/28/2016).
- [33] iOpus. *Browser Scripting, Data Extraction and Web Testing by iMacros*. <http://www.ioopus.com/imacros/>. July 2013.
- [34] iOpus. *Our Products Simplify Website And Web Application Management | iMacros Software*. <https://imacros.net/about/>. Aug. 2019.

- [35] David Karpf. “Social Science Research Methods in Internet Time”. In: *Information, Communication & Society* 15.5 (2012), pp. 639–661. DOI: [10.1080/1369118X.2012.665468](https://doi.org/10.1080/1369118X.2012.665468). eprint: <https://doi.org/10.1080/1369118X.2012.665468>. URL: <https://doi.org/10.1080/1369118X.2012.665468>.
- [36] Salim Khalil and Mohamed Fakir. “RCrawler: An R package for parallel web crawling and scraping”. In: *SoftwareX* 6 (2017), pp. 98–106. ISSN: 2352-7110. DOI: <https://doi.org/10.1016/j.softx.2017.04.004>. URL: <https://www.sciencedirect.com/science/article/pii/S2352711017300110>.
- [37] KimonoLabs. *Kimono: Turn websites into structured APIs from your browser in seconds*. Mar. 2016. URL: <https://www.kimonolabs.com> (visited on 05/08/2015).
- [38] Gary King. “Ensuring the Data-Rich Future of the Social Sciences”. In: *Science* 331.6018 (2011), pp. 719–721. ISSN: 0036-8075. DOI: [10.1126/science.1197872](https://doi.org/10.1126/science.1197872). eprint: <https://science.sciencemag.org/content/331/6018/719.full.pdf>. URL: <https://science.sciencemag.org/content/331/6018/719>.
- [39] Andhy Koesnandar et al. “Using Assertions to Help End-user Programmers Create Dependable Web Macros”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 124–134. ISBN: 978-1-59593-995-1. DOI: [10.1145/1453101.1453119](https://doi.org/10.1145/1453101.1453119). URL: <http://doi.acm.org/10.1145/1453101.1453119>.
- [40] Milind Vidyadhar Kulkarni. “The Galois System: Optimistic Parallelization of Irregular Programs”. AAI3339792. PhD thesis. Ithaca, NY, USA, 2008. ISBN: 978-0-549-96812-2.
- [41] Milind Kulkarni et al. “How Much Parallelism is There in Irregular Applications?” In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’09. Raleigh, NC, USA: ACM, 2009, pp. 3–14. ISBN: 978-1-60558-397-6. DOI: [10.1145/1504176.1504181](https://doi.org/10.1145/1504176.1504181). URL: <http://doi.acm.org/10.1145/1504176.1504181>.
- [42] Nicholas Kushmerick. “Wrapper induction: Efficiency and expressiveness”. In: *Artificial Intelligence* 118.1 (2000), pp. 15–68. ISSN: 0004-3702. DOI: [http://dx.doi.org/10.1016/S0004-3702\(99\)00100-9](http://dx.doi.org/10.1016/S0004-3702(99)00100-9). URL: <http://www.sciencedirect.com/science/article/pii/S0004370299001009>.
- [43] Nicholas Kushmerick. “Wrapper Induction for Information Extraction”. AAI9819266. PhD thesis. 1997. ISBN: 0-591-70843-4.
- [44] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. “Wrapper Induction for Information Extraction”. In: *Proc. IJCAI-97*. 1997. URL: <http://citeseer.nj.nec.com/kushmerick97wrapper.html>.
- [45] Duncan Lang. *getURIAsynchronous function | R Documentation*. <https://www.rdocumentation.org/packages/Rcurl/versions/1.95-4.8/topics/getURIAsynchronous>. Nov. 2017.

- [46] Tessa Lau. “Why PBD systems fail: Lessons learned for usable AI”. In: *CHI 2008 Workshop on Usable AI*. Florence, Italy, 2008.
- [47] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. “Version Space Algebra and its Application to Programming by Demonstration”. In: *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 527–534. ISBN: 1-55860-707-2. URL: <http://portal.acm.org/citation.cfm?id=657973>.
- [48] Tessa Lau et al. “Programming by Demonstration Using Version Space Algebra”. In: *Mach. Learn.* 53.1-2 (Oct. 2003), pp. 111–156. ISSN: 0885-6125. DOI: [10.1023/A:1025671410623](https://doi.org/10.1023/A:1025671410623). URL: <https://doi.org/10.1023/A:1025671410623>.
- [49] David Lazer et al. “Life in the network: The coming age of computational social science”. In: 323 (Jan. 2009).
- [50] Vu Le and Sumit Gulwani. “FlashExtract: A Framework for Data Extraction by Examples”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 542–553. ISBN: 978-1-4503-2784-8. DOI: [10.1145/2594291.2594333](http://doi.acm.org/10.1145/2594291.2594333). URL: <http://doi.acm.org/10.1145/2594291.2594333>.
- [51] Gilly Leshed et al. “CoScripter: automating & sharing how-to knowledge in the enterprise”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '08. Florence, Italy: ACM, 2008, pp. 1719–1728. DOI: [10.1145/1357054.1357323](http://doi.acm.org/10.1145/1357054.1357323). URL: <http://doi.acm.org/10.1145/1357054.1357323>.
- [52] Ian Li et al. “Here’s What I Did: Sharing and Reusing Web Activity with ActionShot”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: ACM, 2010, pp. 723–732. DOI: [10.1145/1753326.1753432](http://doi.acm.org/10.1145/1753326.1753432). URL: <http://doi.acm.org/10.1145/1753326.1753432>.
- [53] James Lin et al. “End-user programming of mashups with Vegemite”. In: *Proceedings of the 14th international conference on Intelligent user interfaces*. IUI '09. Sanibel Island, Florida, USA: ACM, 2009, pp. 97–106. DOI: [10.1145/1502650.1502667](http://doi.acm.org/10.1145/1502650.1502667). URL: <http://doi.acm.org/10.1145/1502650.1502667>.
- [54] Jalal Mahmud and Tessa Lau. “Lowering the barriers to website testing with CoTester”. In: *Proceedings of the 15th international conference on Intelligent user interfaces*. IUI '10. Hong Kong, China: ACM, 2010, pp. 169–178. DOI: [10.1145/1719970.1719994](http://doi.acm.org/10.1145/1719970.1719994). URL: <http://doi.acm.org/10.1145/1719970.1719994>.
- [55] Lev Manovich. “Trending: The Promises and the Challenges of Big Social Data”. In: (Jan. 2012), pp. 460–475. DOI: [10.5749/minnesota/9780816677948.003.0047](https://doi.org/10.5749/minnesota/9780816677948.003.0047).
- [56] Noortje Marres and Esther Weltevrede. “Scraping the Social?” In: *Journal of Cultural Economy* 6.3 (2013), pp. 313–335. DOI: [10.1080/17530350.2013.772070](https://doi.org/10.1080/17530350.2013.772070), eprint: <https://doi.org/10.1080/17530350.2013.772070>. URL: <https://doi.org/10.1080/17530350.2013.772070>.

- [57] Mikaël Mayer et al. “User Interaction Models for Disambiguation in Programming by Example”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. Daegu, Kyungpook, Republic of Korea: ACM, 2015, pp. 291–301. ISBN: 978-1-4503-3779-3. DOI: [10.1145/2807442.2807459](https://doi.org/10.1145/2807442.2807459). URL: <http://doi.acm.org/10.1145/2807442.2807459>.
- [58] SIMILE Semantic Interoperability of Metadata and Information in unLike Environments. *Solvent*. Mar. 2016. URL: <http://simile.mit.edu/solvent/> (visited on 03/28/2016).
- [59] Mixu. *Modern web applications: an overview*. Jan. 2016. URL: <http://singlepageappbook.com/goal.html>.
- [60] Mozenda. *Web Scraping Solutions for Every Need*. Mar. 2018. URL: https://www.mozenda.com/?utm_source=googleadwords&utm_medium=cpc&utm_term=Mozenda&gclid=EAIaIQobChMIuM71jfGc2gIVC7nACh2m0wz8EAAYASAAEgLSzPD_BwE (visited on 03/28/2018).
- [61] Ion Muslea, Steve Minton, and Craig Knoblock. “A Hierarchical Approach to Wrapper Induction”. In: *Proceedings of the Third Annual Conference on Autonomous Agents*. AGENTS '99. Seattle, Washington, USA: ACM, 1999, pp. 190–197. ISBN: 1-58113-066-X. DOI: [10.1145/301136.301191](https://doi.org/10.1145/301136.301191). URL: <http://doi.acm.org/10.1145/301136.301191>.
- [62] Yang Ni et al. “Open Nesting in Software Transactional Memory”. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '07. San Jose, California, USA: ACM, 2007, pp. 68–78. ISBN: 978-1-59593-602-8. DOI: [10.1145/1229428.1229442](https://doi.org/10.1145/1229428.1229442). URL: <http://doi.acm.org/10.1145/1229428.1229442>.
- [63] Nokogiri. *Tutorials - Nokogiri*. <http://www.nokogiri.org/>. Nov. 2016.
- [64] Donald A. Norman. *The Design of Everyday Things*. New York, NY, USA: Basic Books, Inc., 2002. ISBN: 9780465067107.
- [65] Octoparse. *Web Scraping Tool & Free Web Crawlers for Data Extraction | Octoparse*. Mar. 2018. URL: <https://www.octoparse.com/> (visited on 03/28/2018).
- [66] Adi Omari, Sharon Shoham, and Eran Yahav. “Synthesis of Forgiving Data Extractors”. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. WSDM '17. Cambridge, United Kingdom: ACM, 2017, pp. 385–394. ISBN: 978-1-4503-4675-7. DOI: [10.1145/3018661.3018740](https://doi.org/10.1145/3018661.3018740). URL: <http://doi.acm.org/10.1145/3018661.3018740>.
- [67] ParseHub. *Free web scraping - Download the most powerful web scraper | ParseHub*. Mar. 2018. URL: <https://www.parsehub.com/> (visited on 03/28/2018).
- [68] Keshav Pingali et al. “The Tao of Parallelism in Algorithms”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 12–25. ISSN: 0362-1340. DOI: [10.1145/1993316.1993501](https://doi.org/10.1145/1993316.1993501). URL: <http://doi.acm.org/10.1145/1993316.1993501>.
- [69] Platypus. *Platypus*. <http://platypus.mozdev.org/>. Nov. 2013.

- [70] ONPE - Oficina Nacional de Procesos Electorales. *SEGUNDA ELECCIÓN PRESIDENCIAL 2016: ACTAS POR UBIGEO*. <https://www.web.onpe.gob.pe/modElecciones/elecciones/elecciones2016/PRP2V2016/Actas-por-Ubigeo.html#posicion>. Nov. 2017.
- [71] Amit Rathi. *Architectural Shift in Web Applications*. Jan. 2017. URL: <https://dzone.com/articles/architectural-shift-in-web-applications-with-emerg>.
- [72] Mitchel Resnick et al. "Scratch: Programming for All". In: *Commun. ACM* 52.11 (Nov. 2009), pp. 60–67. ISSN: 0001-0782. DOI: [10.1145/1592761.1592779](https://doi.org/10.1145/1592761.1592779). URL: <http://doi.acm.org/10.1145/1592761.1592779>.
- [73] Leonard Richardson. *Beautiful Soup: We called him Tortoise because he taught us*. <http://www.crummy.com/software/BeautifulSoup/>. Mar. 2016.
- [74] Evelyn Ruppert. "Rethinking empirical social sciences". In: *Dialogues in Human Geography* 3.3 (2013), pp. 268–273. DOI: [10.1177/2043820613514321](https://doi.org/10.1177/2043820613514321), eprint: <https://doi.org/10.1177/2043820613514321>. URL: <https://doi.org/10.1177/2043820613514321>.
- [75] Ido Safruti. "Five Easy Ways To Identify Bot Attacks On Your Site". In: *Forbes* (Feb. 2018). URL: <https://www.forbes.com/sites/forbestechcouncil/2018/02/28/five-easy-ways-to-identify-bot-attacks-on-your-site/#57423e188fa3>.
- [76] M.J. Salganik. *Bit by Bit: Social Research in the Digital Age*. Princeton University Press, 2017. ISBN: 9781400888184. URL: <https://books.google.com/books?id=RqMnDwAAQBAJ>.
- [77] Mike Savage and Roger Burrows. "The Coming Crisis of Empirical Sociology". In: *Sociology* 41.5 (2007), pp. 885–899. DOI: [10.1177/0038038507080443](https://doi.org/10.1177/0038038507080443), eprint: <https://doi.org/10.1177/0038038507080443>. URL: <https://doi.org/10.1177/0038038507080443>.
- [78] C. Scaffidi, M. Shaw, and B. Myers. "Estimating the numbers of end users and end user programmers". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. Sept. 2005, pp. 207–214. DOI: [10.1109/VLHCC.2005.34](https://doi.org/10.1109/VLHCC.2005.34).
- [79] Scrapinghub. *Visual scraping with Portia*. Mar. 2018. URL: <https://scrapinghub.com/portia> (visited on 03/28/2018).
- [80] Scrapy. *Scrapy*. <http://scrapy.org/>. July 2013.
- [81] scrapy-plugins. *scrapy-plugins/scrapy-deltafetch: Scrapy spider middleware to ignore requests to pages containing items seen in previous crawls*. <https://github.com/scrapy-plugins/scrapy-deltafetch>. Mar. 2017. (Visited on 03/30/2017).
- [82] Selenium. *Selenium IDE Plugins*. <http://www.seleniumhq.org/projects/ide/>. Mar. 2016. URL: <http://www.seleniumhq.org/projects/ide/>.
- [83] Selenium. *Selenium-Web Browser Automation*. <http://seleniumhq.org/>. July 2013.
- [84] Armando Solar-Lezama. "Program Synthesis by Sketching". AAI3353225. PhD thesis. Berkeley, CA, USA, 2008. ISBN: 978-1-109-09745-0.

- [85] StackOverflow. *Posts containing “incremental scraping” - Stack Overflow*. 2017. URL: <http://stackoverflow.com/search?q=incremental+scraping> (visited on 03/30/2017).
- [86] Atsushi Sugiura and Yoshiyuki Koseki. “Internet Scrapbook: Creating Personalized World Wide Web Pages”. In: *CHI '97 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '97. Atlanta, Georgia: ACM, 1997, pp. 343–344. ISBN: 0-89791-926-2. DOI: [10.1145/1120212.1120425](https://doi.org/10.1145/1120212.1120425). URL: <http://doi.acm.org/10.1145/1120212.1120425>.
- [87] VisualWebRipper. *Visual Web Ripper | Data Extraction Software*. <http://visualwebripper.com/>. Apr. 2017.
- [88] Hanna Wallach. “Computational Social Science & Computer Science + Social Data”. In: *Commun. ACM* 61.3 (Feb. 2018), pp. 42–44. ISSN: 0001-0782. DOI: [10.1145/3132698](https://doi.org/10.1145/3132698). URL: <http://doi.acm.org/10.1145/3132698>.
- [89] Howard T Welser et al. “Distilling digital traces: Computational social science approaches to studying the internet”. In: *Handbook of online research methods* (2008), pp. 116–140.
- [90] 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* 24.1240842. New York, NY, USA: ACM. DOI: [10.1145/1240624.1240842](https://doi.org/10.1145/1240624.1240842).
- [91] Shuyi Zheng et al. “Efficient Record-level Wrapper Induction”. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM '09. Hong Kong, China: ACM, 2009, pp. 47–56. ISBN: 978-1-60558-512-3. DOI: [10.1145/1645953.1645962](https://doi.org/10.1145/1645953.1645962). URL: <http://doi.acm.org/10.1145/1645953.1645962>.