In the future, anyone will be able to write programs that are currently the exclusive domain of advanced programmers. For now, there is still 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. However, the need is pressing; while there are about 20 million programmers in the world, there are now at least twice as many end users writing code to work with data. My work serves these users. I am a **Programming Systems** researcher, and I join **PL** and **HCI** expertise to develop programming languages and programming tools for non-traditional audiences.

I take on big, real-world problems and solve them with a combination of user-centered design and novel problem reformulations. My thesis work was motivated by social scientists who wanted to scrape data from webpages. After studying their needs, I developed a Programming by Demonstration (PBD) tool, fulfilling the longstanding dream of extending PBD to real-world web automation tasks, an open problem for at least a decade. I solved it by decomposing the problem into subproblems solvable with novel (i) user interaction models, (ii) program synthesis algorithms, and (iii) language constructs. The end result is a mature, usable programming ecosystem that has already contributed to several important social science results.

DORA Programming Tools

When I begin working with a new class of programmers, I do formative studies, observing and listening to the target audience. One key lesson from this user-centered approach is that non-traditional programmers find it easy to edit programs but difficult to write them from scratch. In response to this observation, I have developed novel programming tools around a **Demonstrate Once, Revise Anytime (DORA)** workflow, centering on simple *user interaction models* for producing a draft program by providing demonstrations or examples and usable *languages*, so that users can edit their draft programs (Fig. 1). Throughout this statement, I will emphasize: (i) **usable program drafting tools**, (ii) **learnable programming languages**, and (iii) mingling PL and HCI expertise to **co-design** tools and languages.

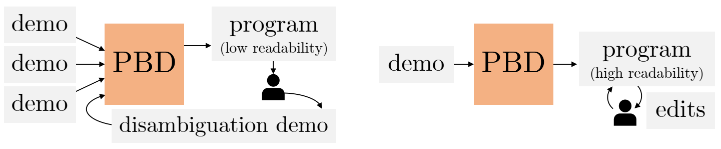

Figure 1: Traditional PBD (left). User provides multiple demos, PBD tool must synthesize perfect program. DORA PBD (right). User provides one demo, PBD tool synthesizes draft program in user-friendly language, user edits program directly.

Traditionally, tool designers build PBD systems around the idea that users cannot edit the synthesizer's output programs. The audience is often non-technical, and the target language is usually low-level, so even trained coders will not edit synthesized programs. The typical outcome is a complicated PBD synthesizer that demands many (often repetitive) demonstrations from the user in its attempt to achieve the perfect program. However, even the most ambitious PBD synthesizer may fail to produce the target program, and if the output program is unreadable, the user has no recourse when PBD fails.

Traditional PBD underutilizes social and data scientists' programming skills. These occasional programmers only feel comfortable *drafting* data analysis scripts, but they feel comfortable *editing* a much broader range of programs. In my DORA style, PBD bootstraps the programming interaction, synthesizing a draft program based on only a single demonstration. The draft alleviates 'blank page syndrome' and initiates a primarily recognition-based programming interaction, a shift away from the recall-based interaction of traditional coding. The PBD tool synthesizes low-level details to shield users from concepts below their level of abstraction, but output programs surface high-level program structure in a learnable language, so users can read and edit them with minimal training. Users customize draft programs to meet their needs, sidestepping the thorny issue of synthesizers that cannot express users' target programs. Users also edit drafts to do complicated programming tasks like parallelization and failure recovery, tasks that cannot be demonstrated but which end users can complete with custom language support. Ultimately, DORA lets the tool designer delegate low-level programming tasks to the machine and high-level reasoning about program goals to the user.

DORA for Web Automation.

The Helena language and its associated PBD tool illustrate my research approach. I started with a high-level goal, to help non-traditional programmers automate repetitive webpage interaction and scraping tasks.

I observed and interviewed social scientists. Then, piece by piece, I built the components I would need to deliver a usable tool (Table 1): at the bottom of the stack, a replayer that is robust to modern interactive webpages, then a language that hides the low-level details required for robust replay, then a PBD tool for writing programs in the language, and finally novel language constructs that support tasks like parallelization that are traditionally out of reach for end users. The end results are: (i) the Helena language, a learnable programming language for web automation tasks, and (ii) the Helena tool, a PBD drafting tool that synthesizes scraping programs based on a user's demonstration of how to collect a small slice of the target dataset from a standard browser. Achieving PBD for web automation required decomposing the problem in new ways, and it required co-evolving these two research outputs. Adjustments in either the PL or HCI domain were fruitless without coordination with the other.

Helena represents the first evidence that Programming by Demonstration (PBD) can be used to automate large programming tasks—tasks that take experienced programmers hours rather than minutes—and it puts these large tasks in range for non-traditional programmers. My social science collaborators have used Helena for projects aimed at: helping low-income families move to high-opportunity neighborhoods; helping charitable foundations reach new audiences; improving the transparency of government agencies; reducing the effects of natural disasters on infrastructure; and much more. I know of at least 20 teams of social scientists using Helena for research, I have published with three of those teams, and I have already been asked to teach Helena workshops.

Building Block	**Domain**	**Details**
Task-focused constructs, *e.g.*, the skip block for reliability and parallelization	PL design	OOPSLA17
Helena PBD tool, a synthesizer for generalization, traversing relations, trees	PBD, interaction design	UIST18
Helena, a web automation language that facilitates generalization	PL design	helena-lang.org
Ringer, a record and replay tool that delays synthesis subtasks for robustness	PBD	OOPSLA16

Table 1: The stack of tools and languages I built to implement my Helena web automation ecosystem.

Drafting: Program Synthesizers. To implement Helena, I developed two synthesizers. The replay synthesizer [1] accepts a straight-line demonstration as input and produces a straight-line replay script as output. A key challenge is that replay scripts must interact with an unstable environment. Webpages are redesigned, repopulated with new content, obfuscated, and A/B tested often, so synthesizing robust node selectors (for finding webpage elements like buttons, links, or textboxes) is a longstanding challenge problem. I reformulated the synthesis task to circumvent this hard problem. Rather than synthesizing a node selector at record-time, I delayed selector synthesis until replay-time, when the tool can leverage information about how the page has actually changed.

The generalization synthesizer [2] accepts as input a straight-line replay script, the output from the replay synthesizer. Its output is a program for scraping or otherwise interacting with large web datasets. My formative studies revealed potential Helena users wanted to collect hierarchical data—*e.g.*, not just a list of authors but a tree of authors and their papers. Traversing hierarchical data requires nested loops, but nested loop PBD was an open problem. The space of allowable programs is huge, and user-provided demonstrations are too ambiguous to narrow the field. I solved this problem via a custom interaction model that makes the synthesis problem tractable. Helena asks the user to demonstrate one iteration of each nested loop (*e.g.*, collect one row of each table in the output dataset) and leverages domain-specific insights to identify objects that should be handled together (*e.g.*, the rows of a given table). With the target tables in hand, Helena applies parameterization-by-value [3] to generalize over table rows. I designed this user interaction model to both (i) offer usable human-synthesizer interaction and (ii) constrain the space of allowable programs to make synthesis tractable.

Editing: Programming Languages. The Helena language helps non-traditional programmers adapt the outputs of synthesis to meet their particular needs and to handle complex programming tasks like parallelization and failure recovery. Helena's skip block offers a concrete example of how language design can support end-user edits. The skip block [4] lets non-coders parallelize and distribute their web automation programs. While many PBD tools are designed to produce single-use, short-running programs, Helena is targeted at collecting large-scale datasets. Thus, like traditional programmers, Helena programmers face performance issues. Since network latency is the primary driver of execution times, parallelization is a natural solution. However, we cannot ask this class of users to reason about parallel algorithms. Instead, the skip block asks users how they would decide whether two rows of a table are duplicates. The key idea is that if a set of attributes is sufficient to conclude that two objects are the same, it can also provide a unique ID for each object. By communicating with a centralized store that tracks which objects have been claimed by each parallel worker, the skip block decides to either skip or execute the block associated with each object it encounters. The user reasons at the high level of the output data, with the result that users who identify as non-coders can parallelize Helena programs in 61 seconds on average.

Other Domains Helena is an excellent illustrative example, but I have applied this same style of research and this same union of PL and HCI expertise to a variety of domains. I have developed DORA workflows for domains ranging from generative Bayesian models of input datasets [5] to reactive robot motion controllers [6] to custom voice assistant skills, and I have begun exploring even more, *e.g.* experiment design and analysis [7].

Future Directions

My discussions with social and data scientists reveal their 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; data storage and schema matching solutions adapted to their needs and expertise. Meeting these diverse programming needs will require co-designing new PL and HCI innovations.

Conversational Programming Environments. For now, my social science collaborators usually complete automatable tasks by teaching research assistants to do them manually. To make programming the easiest alternative, we need editors that demand the same feedback as a motivated research assistant. For instance, say we want an email-based experiment to explore the effects of perceived race on how landlords respond to potential tenants. We could demonstrate how to send a few emails and record their metadata. This demonstration can teach a human assistant or a PBD tool, and a good PBD tool should ask the same questions a human asks. "How many emails should I send, and how often? What if I find a duplicate email address?" For unconstrained tasks, identifying when the specification is ambiguous—and thus when to ask a question—is a challenge in and of itself. Another key challenge is developing programming environments that can integrate knowledge from many sources, from question-answering interactions, demonstrations, and direct program edits. I will treat conversational editor interactions as collaborative specification-building in order to make programming more like teaching a human.

Synthesis from Fuzzy Specifications. For some domains, we need programs for which current synthesis techniques are a bad fit. For instance, if a psychologist wants to test whether a chatbot conversation can alleviate depression, a PBD tool could draft a proof-of-concept chatbot based on a set of user-written input dialogues. The output program should be a probabilistic model of the user's state, based on the history of the user's utterances, and a map from the user's state to the bot's next response. Building probabilistic models is outside the range of standard program synthesis tools, so we cannot rely on traditional synthesis. Since building interpretable models is also outside the range of standard program *learning* tools, we cannot rely on traditional learning to draft these programs either. To produce readable, editable programs from soft specifications, we need new synthesis techniques. I will treat unusual inputs as (fuzzy) specifications to put new domains in range for synthesis.

Non-Traditional Programmers and Big Software. From traditional libraries to DSLs to web-backed APIs, much of modern programming revolves around integrating pre-written modules. As a motivating example, consider a task from one of my sociologist collaborators, extracting latitude and longitude from rental listings. Some listings have a street address field, some only have the address within a text description, some have no address but include location information in a map descriptor, and some have none of these. There are APIs for mapping addresses to latitudes and longitudes, but standard PBD is a bad fit here because the user cannot provide the latitude and longitude of a sample listing. This is common; libraries often do work that users cannot predict well. Instead, we need a new program-authoring interaction that coaches the user through providing a demonstration, offering intermediate results that the user can accept or reject. With Kyle Thayer and Andy Ko, I have begun studying what constitutes robust API knowledge and how to teach it [8]. I will build on this work to develop accessible programming interactions and synthesis techniques for composing existing programs. In particular, I will leverage probabilistic language models—and users' comfort with examples—to build 'demonstrations' incrementally and thus to extend PBD to composition-based domains where standard PBD cannot apply.

Non-Traditional Programmers and Cutting-Edge Software. Sometimes users face tasks with existing solutions—*e.g.*, mapping street addresses to latitudes and longitudes—but sometimes they face unsolved tasks—*e.g.*, classifying Google Street View images as "high pedestrian safety" or "low pedestrian safety." Since training cutting-edge neural networks requires industry- or lab-level resources, we cannot yet bring advanced ML to non-traditional programmers. However, once those tools have been developed by experts, we can put them to work for users outside those elite circles. There has already been work on composing machine-learned building blocks for edit-less contexts. I plan to make this available to non-traditional programmers by treating pre-trained models as a library of functions and applying a DORA workflow to programs for composing them. I expect to reduce the number of labels the user provides at the cost of having a dialogue with the user: Is `crosswalk` $\in$ `objects_recognized` good evidence of "high pedestrian safety"? What about `billboard` $\in$ `objects_recognized`? This approach is applicable to many domains beyond computer vision and, crucially, this PL perspective simplifies the problem to the point where tool design does not require special insights in those domains. By treating expert-written programs as the black-box abstractions of a DSL, composing them via synthesis, and augmenting the specification with users' domain knowledge, I will put cutting-edge technologies in reach for non-traditional programmers.

Vision. Empowering technologies from PL and HCI are about to launch a new era in which we can blur the line between programmers and non-programmers. Making programming accessible is foundational to expanding the impact of CS at large, and I will drive research that advances this goal by making automation as easy as demonstration.

References

[1] Shaon Barman, Sarah E. Chasins, Rastislav Bodik, and Sumit Gulwani. Ringer: Web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 748–764, 2016.

[2] Sarah E. 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, New York, NY, USA, 2018. ACM.

[3] Sarah E. Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. 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, pages 179–182, 2015.

[4] Sarah E. Chasins and Rastislav Bodik. Skip blocks: Reusing execution history to accelerate web scripts. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):51:1–51:28, October 2017.

[5] Sarah E. Chasins and Phitchaya Mangpo Phothilimthana. Data-driven synthesis of full probabilistic programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 279–304, 2017.

[6] Sarah E. Chasins and Julie L. Newcomb. Using SyGuS to synthesize reactive motion plans. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 3–20, 2016.

[7] Eunice Jun, Jared Roesch, and Sarah E. Chasins. Experimental design as programs. In *PNW PLSE*, 2018.

[8] Kyle Thayer, Sarah E. Chasins, and Andrew J. Ko. A theory of robust API knowledge. *ACM Transactions on Computing Education*, forthcoming.