Drug trial data analyses, climate change and climate economics models, programs for detecting gerrymandering—these programs shape our lives and our neighbors' lives. The domain experts who write these programs are practitioners with deep domain expertise but no formal computing education. These social scientists, scientists, journalists, policymakers, and other non-traditional programmers work in high-stakes settings where bugs aren't about run time or downtime. A bug is the difference between helping and harming whole communities.

My lab develops novel programming tools by studying domain experts who work on high-stakes or societally important questions. These populations have vastly different needs and a diverse range of skillsets, in contrast to the comparatively homogeneous population of software engineers supported with mainstream programming languages and tools. It is this diversity of needs and skillsets—and the risks of ignoring it—that drive my lab's work. Our work is not ultimately driven by convenience or usability, although those can play a role. Instead, the core of our research is to develop novel programming interactions that *don't lead users astray*.

The "secret sauce" behind all of my lab's research is: (i) Close collaboration with non-traditional programmers from a diverse set of fields. (ii) Combining techniques from Programming Languages (PL) and Human-Computer Interaction (HCI). Together, these have produced lessons that guide our work across a range of tools. In this statement, I'll highlight two recurring lessons in particular: Center editing rather than drafting. Let practitioners express constraints both on program structure and on program output.

If you're interested in reading about a few of the key themes running through my lab's research, continue to Section 1. If you'd like to jump straight to reading about a couple recent projects that illustrate those themes, go ahead and skip to Section 2.

1 Building Better Programming Tools by Understanding Programmers

1.1 Edit-Centric Programming: Helping Users Navigate the Space of Possible Programs

As we build more and more programming tools for domain experts, it has become obvious: even non-programmers spend more time editing, adapting, and maintaining programs than they spend on writing a first version [14, 16, 4]. As a community, we've built a strong foundation of tools for drafting programs from scratch. Now we're ready to introduce the next generation of tools that will build on that foundation, wrestling with the direction-changing observation that even our best program drafting tools only work if users can understand and edit the generated programs.

The need to support program editing imposes difficult constraints. For example, the program must be expressed in a language the user can read, and the authoring interaction must support the user in building a mental model of how their program works. The recent surge in LLM-authored code has underscored the importance of this point. Many developers now describe the difficulty of understanding and modifying codebases in which they or others have "vibe coded." We also see a rising recognition of the fact that reading and editing code is harder than generating it, even for professional programmers, and that this pattern is connected to the sudden influx of LLM-inserted bugs in deployed software products.

From my lab's most recent work on explicitly navigating between neighboring programs [9] all the way back to my own thesis work on DORA workflows (Demonstrate Once, Refine Anytime), my lab's program authoring tools have always emphasized the importance of *editing and refinement* over one-shot drafting. The central importance of editing informs all of our work (for example, by constraining us to user-understandable program representations), but it sits at the heart of a few. Our work and others' have revealed that synthesizer- and LLM-generated code becomes much less useful if programmers don't understand it enough to edit it [5]. We've learned that the ability to tweak and adapt programs is highly prized across many domains [16, 14, 5, 11]. We've learned how editing behavior depends on language characteristics [7, 14]. We've used analyses of the particular editing actions that real users make—and the bugs that appear when they make them—to guide language design interventions [12, 11].

As we've realized the central importance of editing, we've designed programming tools specifically to support the editing journey. Our recent tools explore how programmers benefit from understanding where they can go next from a given program—understanding the neighborhood of 'adjacent' programs. For instance, we introduced a new program synthesis technique, Programming By Navigation (PBN), around the idea of explicitly navigating the space *of only valid programs* [9]. To achieve this, we must know enough about the programmer's intent to prune all invalid paths through the program space. After that, the key research questions are about how to support the user in stepping between valid nodes. (See Sec 2.) Another recent work centers on formalizing the effects of all allowable GUI edit actions in a visual programming environment; we give semantics to all possible program diffs, representing each diff's effects both on the program text and on the program output [17]. For settings where the programmer doesn't know what edit to make and thus can't use a GUI to make it, there's Programming By Scaffolded Demonstration (PBSD). A PBSD tool depicts a variety of small program edits, but without acquiring a definition of validity in advance (in contrast to PBN above). Instead the user must assess whether a given intermediate value moves them closer to their target output. We can think of PBSD as a way of achieving Programming By Demonstration even when the user doesn't know the final output and can't demonstrate how to get there. Instead the user picks a path through the large, branching graph of possible edits. The PBSD tool preemptively visualizes the nearby part of the space, showing how each AST transformation changes both the program and the output. We can apply the program navigation idea at various levels of granularity, from tiny character-level ed-

its to the sets of edits that programmers themselves find meaningful, as in our work on building up (i) the tree of nearby program outputs and (ii) how the user traveled between them, based on the user committing to version control [13]. Our tool supported users in organizing the program space in a way that aligned with how they'd explored it, changed how they visited and revisited program variants, and changed the programs they ended up with, specifically by changing how they explored the program space.

Our emphasis on editing comes in part from the observation that editing is a harder, bigger, longer-running portion of the programming process than drafting. However, this also connects to our top-priority goal of not leading users astray. Jumping directly to a "complete" program can seem convenient. However, a drafting-only process can lock users into programs they didn't actually want [5]. A programming tool that treats editing as a first-class concern means there's still a path to reach the right program [4], even if drafting didn't go as planned. Even better, we can offer edit-centric styles of program drafting, which make users wrestle with more of the key decisions—bringing them directly to the program they actually want [9, 17].

1.2 Sometimes Users Want to Edit the Program; Sometimes Users Want to Edit the Output

Some edits are easy to make directly on a program, reasoning about program structure. Other edits are easy to make indirectly, by manipulating the program output and letting the change propagate back to the program [5, 16, 11]. The more we build tools that force users to make only program edits or only output edits, the more we realize that users want the choice to do either.

My lab explores how the programming process changes when we treat the choice between direct program edits and indirect output edits as a spectrum. In fact, we can build programming tools that move users along two separate but interacting spectrums: (1) Does the user edit the program mostly by editing the program text or program structure directly, or mostly by editing the program output? (2) Does the user spend the majority of their time thinking about the code or the program structure, or do they spend the majority of their time thinking about the program output? Figure 1 shows a pictorial representation of the space defined by these two axes, indicating where a few of my lab's programming tools sit on these axes.

In Figure 1, traditional textual editing appears at the lower left—the programmer thinks about the program structure and edits the program text. In the synthesis literature, we have built a solid foundation of works that move programmers towards thinking mostly or only about outputs. Traditional Programming By Demonstration (PBD) and Programming By Example (PBE) ask the programmer to think about the program output and edit the program via output edits. These tools would sit right at the upper right in Figure 1. Historically, we have tended to assume that tools closer to the upper right are automatically more usable (and therefore better), across the board. In fact, some edits are straightforward to make on one side of the spectrum and some on the other, as our work has revealed repeatedly [5, 11]. Likewise, some parts of the programming process are easier if the user is mostly thinking about the program structure and some if they are mostly thinking about output [16]. In short, we may find compelling programming interactions anywhere in the whole two-dimensional space.

For a concrete illustration, we'll take a whirlwind tour of two tools that inhabit underexplored positions in the space. With Quickpose [13], the programmer exclusively makes edits textually, in the same way as with traditional programming. However, the programming environment uses representations of the program outputs to focus users on how they've evolved the output and the connections between codebase versions in terms of those outputs; users therefore spend much of their time thinking not about the program structure but about the edges in a tree of evolving program outputs. A user study revealed this supported them in tasks ranging from understanding program context to backtracking to coming up with new ideas. In another tool, Perpend, the user sees a large set of tool-proposed edits both in terms of the effect on the program and in terms of the effect on the output. Our study revealed this changes programmers' behavior relative to straightforward DM programming, which requires users to have a particular program output in mind first, and to implement it via GUI interactions in order to take the next step forward. Specifically, it helped them make progress both when their primary intention for their program was focused on the program and its structure *and* when their primary intention for their program was focused on its behavior and output. They transitioned flexibly between modes throughout the programming process, in contrast to the DM tool which enforced that edits start from a specific output manipulation.

Recall our focus on programming tools that don't lead programmers astray. We emphasize tooling that keeps users informed about the meaning of the current program and accepts all inputs users can share. Some lessons about program behavior are easier to learn from reading the program, while others are easier to learn from seeing an execution or an output; this drives our focus on letting users move flexibly across the first axis. Some corrections to program behavior are easier to express in the program text, while others are easier to demonstrate by manipulating an output; this drives our focus on letting users move flexibly across the second axis. It is the programming tool's responsibility to keep the user informed so the user can spot—and ultimately correct—divergence from their intended program behavior.

1.3 Using Human-Centered Work to Inform Novel PL

Outside of my own lab's research, my most important role is to support the growing research subfield that sits at the intersection of PL and HCI. In particular, since 2017 I am one of three co-organizers of PLATEAU, the biggest venue at the intersection of PL and HCI. With co-chairs Joshua Sunshine and Elena Glassman, we've taken PLATEAU free-standing over the past four years, where previously it was co-located either with a top PL or top HCI conference, depending on the

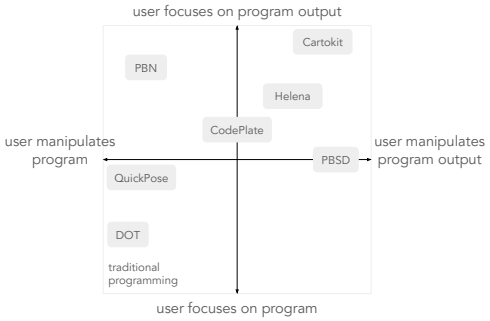

Figure 1: My lab is engaged in a long-term project of exploring the effectiveness of programming interactions that vary on two key axes: (1) Edit Actions: Does the user manipulate the program by directly manipulating the program or indirectly, by manipulating the program output? (X-axis at left.) (2) Programmer Focus: Does the user focus on the program or on the program output? (Y-axis at left.) Traditional programming processes appear at the lower left. The top half of this space remains dramatically underexplored—especially the portions of the space where the programmer can manipulate *both* the output *and* the program itself. Roughly, we can think of this top half of the space as corresponding to programming interactions in which the programmer spends a minority of their time thinking about the program structure. The figure illustrates where a selection of my lab's programming tools sit on these two axes, including [13] described here and [9, 17] described in Sec 2.

year. Our first independent year was 2021, and we've now held well-attended independent PLATEAUs in a range of locations, including one here on the Berkeley campus. We've made many improvements to PLATEAU, mostly centered around growing the community and improving mentoring. We've seen a spike in PL-HCI hiring over the last few years. These new junior faculty interacted with PLATEAU en route to becoming faculty, and most continue to engage.

The tide of PL-HCI research continues to rise, and in particular the PL community is adopting more and more HCI techniques. Many, many researchers' works and efforts are driving this evolution. I am happy to hear from junior and senior colleagues alike that my lab's work is playing a role. Sometimes junior researchers point to the works one might expect; e.g., "PL and HCI, Better Together" [1] explicitly lays out the case for how PL can benefit from HCI insights and techniques. But often they point to less expected papers; for instance, many researchers tell me reading "How Statically Typed Functional Programmers Write Code" [7] was the experience that persuaded them they needed to integrate HCI work into their PL research practice.

To illustrate a concrete change in the PL community, consider the new prevalence of *need-finding research*. When a PhD student enters my lab, we often start by building an understanding of the needs of a particular programming audience. This kind of research is typically called need finding in HCI, and—until recently—it tended to be rare in PL. We use need-finding work to shape our choices about the tools that we build for a given audience of coders or non-coders. We have conducted need-finding work to understand: users who work with geospatial data, with a special focus on geographers, data journalists, and social scientists [16]; climate economists and others who work with climate economics models [14]; novice synthesizer users [5]; statically typed functional programming experts [7]; knowledge network content creators who use low-resourced languages [10]; stakeholders in the criminal justice space, including lawyers and journalists [11]; and sociologists [4]. In some cases, the population is so understudied that we need a large-scale need-finding study to build enough knowledge to start selecting research problems. In these cases, we typically publish a free-standing need-finding work. In other cases, when we start with a base of knowledge about the population in question, we can conduct smaller need-finding studies, which we often integrate as a small component of a primarily tool-focused paper. In all cases, we use the need-finding work to inform our choice of research problem and the design of our programming tools.

Importantly, my lab has uncovered our most interesting research problem statements by using HCI techniques. Both of the two preceding themes (edit-centric programming; output- vs. structure-guided programming) arose from user studies. Many of our user study findings are specific to a given setting—e.g., see Section 2.1 for a research challenge we uncovered only via long-running collaboration with experimental biologists. However, my lab also benefits from running so many studies with so many very different audiences. We can identify the patterns that occur repeatedly across audiences, regardless of participants' disciplinary backgrounds and styles of computing education. Need-finding, formative, and evaluative user studies alike have reinforced the importance of the lessons from Sections 1.1 and 1.2. We see these same themes arising, again and again, across a diverse array of disciplines and programming tasks.

1.4 Impact Outside of Computing

My lab is deeply committed to impact on domains outside of computing. So far, we have released *new* programming tools for domain experts including: geographers, cartographers, and data journalists [17]; biologists [9, 8]; attorneys and journalists [11]; artists [13]; and sociologists [4]. We have also proposed directions for improving existing programming tools for data scientists [12] and climate economists [14]. Because of our close collaborations with domain experts outside of computing, we also publish with domain experts, both inside computing [11, 4] and outside computing—e.g., recently with our Biology collaborators in *Nature Communications* [15] and various Sociology venues [3, 2, 6]. My lab also builds programming tools for non-academic audiences. For example, one of the tools described in Section 2 has been adopted by data journalists at several national newsrooms.

2 Two Case Studies: Analyzing and Visualizing Wet-Lab Biology Data & Authoring Interactive Maps

For concrete illustrations of the research themes, I'll describe two projects in more detail.

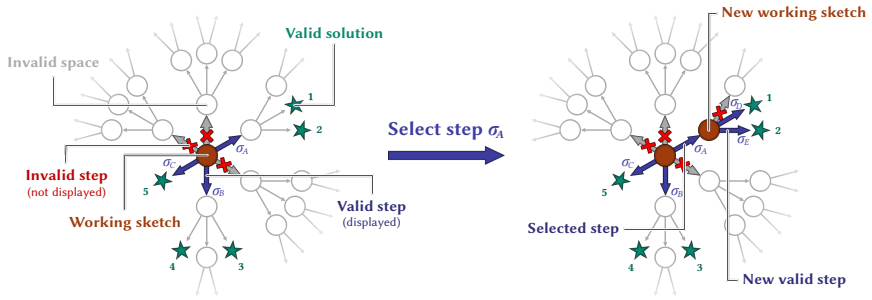

Figure 2: **Two rounds of a Programming by Navigation Interaction. Round 1:** At each round of synthesis, we have a sketch (an in-progress program, depicted as an orange circle), which is initially empty. In this scenario, there are 5 valid synthesis solutions (depicted as green stars). Given a sketch, a PBN synthesizer must return all steps on paths that lead to valid solutions (depicted as purple arrows, annotated σ_A, σ_B, and σ_C), a property we call Strong Completeness. In addition, the synthesizer must not return any steps that cannot lead to a valid solution (depicted as arrows with red ×s), a property we call Strong Soundness. **Round 2:** We depict the result of selecting the particular step σ_A (although the others would have been valid to select as well). The step σ_A gets applied to the previous sketch, resulting in a new sketch, and the PBN synthesizer must again return all and only the valid next steps; now σ_D leads to Solution 1 and σ_E leads to Solution 2.

2.1 Biology Case Study: An Interactive Program Synthesizer That Doesn't Let You Go Wrong

During my student's two years of embedding with a team of wet-lab biologists, we realized two things: (1) Any program authoring tool for our biology collaborators must prevent them from taking a 'wrong step' at every point during program creation. (2) Any program authoring tool must preemptively inform them of all possible 'correct steps,' lest they lack knowledge of the full set of computational biology algorithms available for analyzing their data. No such program authoring tool existed, so we set out to invent one.

Unfortunately for the creators of program synthesis tools, real programmers usually start the programming process with an underspecification—that is, an ambiguous specification. If they use a synthesis tool, this means they usually engage in a back and forth with the synthesizer, iteratively refining their specification until they're happy with the program they get back. This is common knowledge, but most existing synthesis work treats the synthesis process as a one-shot interaction in which the user provides the correct specification, and the synthesizer runs once. In particular, most synthesis works only provide soundness and completeness guarantees about this one-shot execution, not about the interaction as a whole. A few prior works offer guarantees about the iterative specification refinement process—but they do it, not by making guarantees about how the synthesizer will behave, but by making *assumptions* about how the user will behave.

We introduced the Programming by Navigation (PBN) Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution even when the user starts with an underspecification. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). In short, this is a program synthesis tool that *doesn't let the programmer mess up*! There is no way to use a PBN synthesis tool to reach an invalid program or to waste time going down a path on which no valid programs are accessible. (See Figure 2.) This eliminates important classes of bad interactions, not by making assumptions about users, but by ensuring that the tool itself only offers 'good paths.' In particular, it is impossible for the user to go down rabbit holes or to make an initially valid specification into an invalid specification. As an analogy, we can think of a PBN synthesizer as serving a similar role to a structure editor or projectional editor; but where a structure editor prevents users from taking *syntactically* invalid steps, PBN program authoring prevents users from taking *semantically* invalid steps.

At first glance, it may sound as though the only way to implement a PBN synthesizer is by identifying all valid programs in advance. In fact, this approach is impossible, because there may be infinite valid programs. Instead, we introduced an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We defined such an oracle via sound compilation to Datalog. This technique produced an efficient PBN synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer was the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**—that is, it meets requirements (1) and (2) that we identified through my student's work with biologists!

This technique is exactly the innovation we needed to meet the biologists' needs—but also turns out to be broadly applicable. With a library of allowable building blocks, essentially a set of compbio algorithms, our PBN implementation generates Python scripts for analyzing or visualizing data from wet-lab biology experiments. We can apply the same synthesis framework, and even the same implementation, to another domain by simply providing a different library of allowable building blocks.

This work started with the heaviest-weight need-finding process I've ever seen from a PL student, Justin Lubin's deep two-year

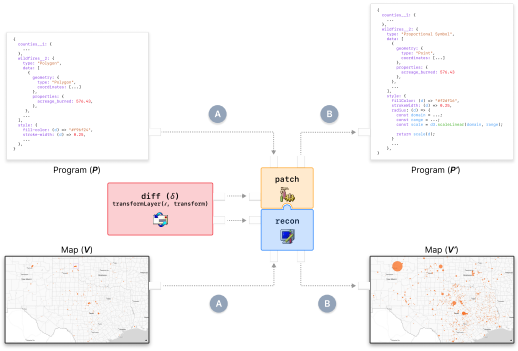

Figure 3: **The `patch-recon` approach applied to a geospatial visualization program.** (A) A GUI interaction creates a `diff`. The system uses the `diff` for (1) the `patch` operation, which receives two inputs, the `diff` and the current program P and (2) the `recon` operation, which also receives two inputs, the `diff` and the current output value V. (B) When `patch` applies the `diff` to program P, it produces the updated program P'. When `recon` uses the `diff` to update V, it produces the updated output value, V'. The program is never forward evaluated to produce the updated output map, even when the map representation in V is quite different from the map representation in V'—as in this example, which shows a transition from using a polygon mark to a proportional symbol mark to represent a given dataset.

embedding in James Nuñez's biology lab. So far, this long-term collaboration with biologists has produced two novel synthesis techniques, the PBN technique described above and presented at PLDI 2025 [9], and a technique based on canonicalization and creatively reusing the long history of compiler optimizations, presented at PLDI 2024 [8]. It has also resulted in a joint publication with our biology collaborators in Nature Communications [15].

Themes This project connects with all four research themes: **Navigating the Space of Programs.** This is the core of Programming By Navigation, and of course explains the name of our technique. The synthesis tool provides the steps by which the user can navigate the space, the program edit actions that link the current sketch to the adjacent nodes in the program space. The user navigates by picking the specific step from the allowable options. **Program vs. Program Output.** Recall that the PBN infrastructure bears the responsibility of checking for the validity of any composition of building blocks. The programmer picks which building blocks they like, but they never have to think about how to integrate them, about the program structure or control flow. Instead the user states facts about the input—e.g., how their data was collected from a wet-lab experiment—and facts about the output—e.g., the kind of analysis that should result. The end result is a programming tool that nudges users to reason about outputs *even though it often takes weeks to run a wet-lab analysis and find its output.* **HCI Techniques in PL.** We invented the Strong Soundness and Strong Completeness guarantees after realizing that these were the specific guarantees we'd need to make biologists successful. Just as all prior interactive synthesis works hadn't delivered Strong Soundness and Strong Completeness, we probably would have overlooked the need for these guarantees if we had not conducted the necessary need-finding work in advance. We decided to pursue Strong Soundness based on the need-finding work, but we can only study its impact on users once we've invented a tool that achieves the guarantee. In the next stage of this research, we will study whether Strong Soundness, which no prior program synthesis tool has ever provided, will benefit users as we predict. The analogous questions for structure editors, about the impact of restricting programmers to syntactically valid programs, has already been studied extensively for over a decade, across dozens of user studies. However, because there has never before been a synthesis tool that guarantees Strong Soundness, this question has never been studied for semantic validity. **Impact Outside Computing.** Finally, our PBN tool is being adopted in UC Berkeley and UCSF, and we will teach it at upcoming events at the San Francisco CZ Biohub and other venues.

2.2 Cartography Case Study: Direct Manipulation Programming For Big, Data-Intensive Programs

Many users continue to prefer Direct Manipulation (DM) interfaces as their primary way of interacting with computers. Although there are few DM *programming* tools, most of the applications and webpages we use in our day-to-day computer usage are DM—if it's not happening at the command line or in a text editor, it's probably DM. Clearly, DM GUIs are much more mainstream and familiar than any program generation tools.

On the surface, it's alluring to think we could give users a DM GUI for building up a concrete value, then automatically generate a program that produces that concrete value as output. This would give users a way to write programs without directly writing code, by using the familiar GUI-style interactions they know from mainstream direct manipulation interfaces. The stumbling block appears when we try to apply this strategy to the long-running programs that real users want. To date, there are a few DM programming systems, all of which use two main components: (1) a *patch* component, which modifies the program based on a GUI interaction, and (2) a *forward evaluator*, which evaluates the modified program to produce an updated program output. This architecture works for developing short-running programs—programs that reliably execute in <1 second—generating outputs such as SVG and HTML documents. However, forward evaluating longer-running programs such as data visualizations in response to every GUI interaction would mean crossing outside of interactive speeds. We extended DM programming to long-running programs by pairing a standard *patch* component with a corresponding *reconciliation* component, `recon`. `recon` directly updates the program *output* in response to a GUI interaction, eliminating the need for forward evaluation (Figure 3). `recon` can operate both incrementally and in parallel with `patch`. Empirically, the longer a program takes to run with forward evaluation, the greater the speedup we see from using our `patch-recon` approach instead. This architecture is easier for developers to reason about (and prove correct) than techniques based on cache invalidation. This work offered the first pathway for extending

direct manipulation programming to domains involving large-scale computation. Now that our `patch-recon` strategy extends DM interfaces to desirable, long-running programs, and given that we know users are comfortable using DM GUIs in many domains, adding program authoring into familiar, existing DM GUIs is a promising route for truly democratizing programming.

As part of this project, my student Parker Ziegler has been working for a year with Grist, a news outlet focused on climate change reporting. He has co-won a journalism award for his work there. However, the impact is not limited to our direct collaborators. Recently, he attended a data journalism conference, NICAR. He learned that news outlets including The New York Times, The Washington Post, and Financial Times had independently found and used Cartokit, our `patch-recon` tool for programming interactive maps.

This work started with need-finding research studying Earth and climate scientists, social scientists, and data journalists, presented at CHI 23 [16]; progressed to introducing the `patch-recon` DM programming approach presented at PLDI 25 [17]; and has now proceeded to producing a suite of related cartography tools including a pure-DM tool and a hybrid DM-LLM tool, in which we can trigger `recon` either with GUI interactions or with natural language descriptions of the desired change.

Themes This project connects with all the research themes highlighted above: **Navigating the Space of Programs.** The `diff`s that we feed to `patch` and `recon` are the key object of study here. While it is standard to apply `diff`s to program text, `recon` is a way to give *semantics* to program `diff`s. `patch-recon` turns the process of engineering a DM programming environment into essentially an exercise in deciding the set of allowable `diff`s, the set of allowable program edits, by which the programmer will be allowed to navigate the space of programs. Notably, the only reason a `patch-recon` architecture improves performance relative to forward evaluation is *because* the DM programming process proceeds via a sequence of program edits, with opportunities to avoid redundant computation. **Program vs. Program Output.** The other core appeal of DM interactions—other than the focus on building outputs via incremental output edits—is the focus on building outputs in general. From the start, the goal of a DM programming system is to give the programmer the flexibility to make edits either by altering the program directly or by altering the output to indirectly edit the program. **HCI Techniques in PL.** Our need-finding work with users of Geospatial Information Systems and geospatial analysis libraries [17] was the driving force behind Cartokit. Our results indicated that practitioners who used only DM (non-programming) tools struggled with tasks that programs could solve for them—e.g., tediously repeating the same mapping process for multiple datasets. In contrast, practitioners who used only programming tools struggled with tasks that DM tools could solve for them—e.g., rapid exploration of multiple different cartographic representations. In fact, we even found that many users already switched back and forth between DM and programming tools in their processes, even though they had to completely throw away their DM result to use programming or throw away their program to use DM. **Impact Outside Computing.** As described above, even though Cartokit is brand new, it has already been organically discovered by data journalists at national newsrooms including the New York Times and the Washington Post, as well as interactive mapmakers outside of journalism.

2.3 The Future of Programming

I'll conclude with closing thoughts on how my lab's work connects to the future of programming.

On AI for Non-Programmers Generative AI has become an important tool in our toolbox for building programming aides. (E.g., see Sec 2.) In isolation, generative AI supports non-traditional programmers in producing code in limited settings: roughly, the same situations in which they could cobble the program together by Googling, plus the situations most similar to those. In other settings, especially for difficult domain-specific code, LLMs alone are a poor fit for non-programmers because: 1. **Problem decomposition is hard** for non-programmers, novice programmers, and LLMs alike. 2. **Coders and non-coders use different concepts and words in natural language**. In an LLM setting, since non-programmers' text doesn't appear alongside code in LLM training sets, their natural language often doesn't produce useful results from LLMs. 3. **Non-programmers may not be able to read traditional programs**. Thus, they may not be able to identify whether LLM-produced code does what they want. 4. **Identifying a next step is hard**. Even if a user can understand their LLM-produced code enough to know it's wrong, they often don't know how to tweak the code, the prompt, or other inputs in ways that will force their desired behavior. 5. **Deciding what they want is hard**. Put another way, navigating the space of specifications is hard. Programmers (often) find it easy to produce code once they have a full specification in mind. Much harder is reaching the point of having a full specification; much of the programming process is about refining the specification, the programmer figuring out what they want. User studies show LLMs make both programmers and non-programmers worse at this process. • The five challenges above are exactly the challenges that programming tools *for domain experts* have been supporting from the start, long before LLMs arrived on the scene. These were always the parts of the programming process that were hardest, and always the reason why we've had to design custom abstractions, environments, and drafting tools for non-programmer domain experts. In short, LLMs have provided a very useful building block for many new programming tools, but surprisingly the fundamental challenges of navigating non-coders to difficult programs have remained remarkably stable—even as LLMs have triggered a massive shift away from search engines and towards LLM-backed tools. With the tremendous surge of interest in LLM-backed programming, I'm optimistic that the community will make *much* faster progress on these five challenges. Our work for domain experts will benefit from the insights this upswell of interest will generate—and,

in turn, our growing body of work offers important lessons for LLM-backed tools on how to overcome these challenges.

Domain Experts and the Future of Programming Programs that shape our daily lives—setting price thresholds for housing voucher programs, the data analysis behind the next medical breakthrough CRISPR-based therapy—are being written by non-programmers. Domain experts are writing programs that can help or harm whole communities. With programs only a chat away, code feels more accessible to more populations than ever before. Whether this trend results in more harm or more good will come down to one question: How well do programming tools support real domain experts in producing correct programs? Without formal computing educations, practitioners need to write, edit, understand, and debug a wide range of deeply difficult programs. We cannot assume a four-year CS education that teaches them the same concepts, skills, and background knowledge that software engineers share. Likewise, we cannot assume the trivial, prepackaged programming tasks that so many expert-targeted tools try to enforce. Our next generation of programming tools must be designed with a deep understanding of domain experts' real programming practices—and how our tools can help them or lead them astray. My lab's work shows that when we design programming tools from the start to meet practitioners' needs, we find fascinating new research problem statements. But more importantly, we invent tools that make domain experts more capable and more correct. At its best, this work isn't about individual productivity or ease, but about expanding the collective problem-solving capabilities of whole disciplines. With this perspective, PL for domain experts sits right at the core of the future of programming tools. The PL work happening right now will determine whether the next generation sees programming as a path to a more informed and evidence-driven society—or just a way to get wrong answers faster.

References

[1] Sarah E. Chasins, Elena Glassman, and Joshua Sunshine. PL and HCI: Better together. In *Communications of the ACM*, CACM '21, 2021.

[2] Ana Costa, Victoria Sass, Ian Kennedy, Roshni Roy, Rebecca J. Walter, Arthur Acolin, Kyle Crowder, Chris Hess, Alex Ramiller, and Sarah E. Chasins. Toward a cross-platform framework: Assessing the comprehensiveness of online rental listings. In *Cityscape Volume 23, No. 2 (HUD Office of Policy Development and Research)*, Cityscape '21, 2021.

[3] Chris Hess, Arthur Acolin, Rebecca Walter, Ian Kennedy, Sarah E. Chasins, and Kyle Crowder. Searching for housing in the digital age: Neighborhood representation on internet rental housing platforms across space, platform and metropolitan segregation. In *Environment and Planning A: Economy and Space*, 2021.

[4] Chris Hess and Sarah E. Chasins. Informing housing policy through web automation: Lessons for designing programming tools for domain experts. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, 2022.

[5] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. Exploring the learnability of program synthesizers by novice programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, 2022.

[6] Ian Kennedy, Amandalynne Paullada, Chris Hess, and Sarah E. Chasins. Racialized discourse in seattle rental ad texts. In *Social Forces, Volume 99, Issue 4*, Social Forces '20, 2020.

[7] Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. In *Proceedings of the 2021 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '21, 2021.

[8] Justin Lubin, Jeremy Ferguson, Kevin Ye, Jacob Yim, and Sarah E. Chasins. Equivalence by canonicalization for synthesis-backed refactoring. In *Proceedings of the 45th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '24, 2024.

[9] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. Programming by navigation. In *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '25, 2025.

[10] Hellina Hailu Nigatu, John Canny, and Sarah E. Chasins. Low-resourced languages and online knowledge repositories: A need-finding study. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, 2024.

[11] Hellina Hailu Nigatu, Lisa Pickoff-White, John Canny, and Sarah Chasins. Co-designing for transparency: Lessons from building a document organization tool in the criminal justice domain. In *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '23, 2023.

[12] Eric Rawn and Sarah E. Chasins. Pagebreaks: Multi-cell scopes in computational notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, 2025.

[13] Eric Rawn, Jingyi Li, Eric Paulos, and Sarah E. Chasins. Understanding version control as material interaction with quickpose. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, 2023.

[14] Lisa Rennels and Sarah E. Chasins. How domain experts use an embedded DSL. In *Proceedings of the 2023 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '23, 2023.

[15] Da Xu, Swen Besselink, Gokul N. Ramadoss, Philip H. Dierks, Justin P. Lubin, Rithu K. Pattali, Jinna I. Brim, Anna E. Christenson, Peter J. Colias, Izaiah J. Ornelas, Carolyn D. Nguyen, Sarah E. Chasins, Bruce R. Conklin, and James K. Nuñez. Programmable epigenome editing by transient delivery of CRISPR epigenome editor ribonucleoproteins. In *Nature Communications*, Nature Communications, 2025.

[16] Parker Ziegler and Sarah E. Chasins. A need-finding study with users of geospatial data. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, 2023.

[17] Parker Ziegler, Justin Lubin, and Sarah E. Chasins. Fast direct manipulation programming with patch-reconciliation correspondence. In *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '25, 2025.