Research Statement

Chenglong Wang
University of Washington

I work on empowering data scientists to solve complex programming tasks. Specifically, I apply program synthesis techniques to build tools that help data scientists solve challenging data manipulation and visualization tasks without programming. Data manipulation and visualization support data scientists' efforts to explore and understand data throughout the analysis process (e.g., to detect outliers during data collection and to conceptualize models during statistical analysis), since such exploratory analyses are their step stones for more complex analysis results and insights. While experienced data scientists can often achieve efficient, flexible and reusable exploratory analysis using programming systems like SQL and R, inexperienced users often struggle to achieve similar results using only interactive tools. My research builds synthesis-powered tools – tools that synthesize programs from examples or partial task specifications – to bridge the gap between interactive data analysis tools (like Excel, Tableau) and programming systems (SQL, R) so that data scientists can solve complex exploratory data analysis tasks they cannot easily solve otherwise. My long-term goal is to democratize programming with program synthesis to allow data scientists to achieve all analysis tasks they can do today only with programming systems.

My dissertation research solves the following challenges data scientists often encounter during exploratory data analysis:

- Many data manipulation tasks involve querying databases using advanced SQL features, like aggregation, subqueries and outer-joins, that many data scientists lack knowledge about. To allow easy access to relational databases, I developed Scythe [11], a SQL query synthesizer that synthesizes queries from input-output examples, a common medium Stack Overflow users use to describe questions (Figure 1.1). To enable Scythe to efficiently explore the combinatorial search space, I developed an abstraction of SQL that helps Scythe reason about realizability of user specifications in parts of the search space and dramatically prune infeasible ones. When tested on a collection of 193 real-world tasks from Stack Overflow, Scythe solved 70% of the tasks (compared to 48% which previous algorithms support). Scythe was also 52$\times$ faster on average for tasks that previous algorithms solved.
- While Scythe makes data manipulation easy, creating expressive visualizations also requires data scientists to know a considerable amount about plotting tools to implement their designs and drive data transformation. To empower users who had neither data transformation or visualization experience to create expressive visualizations, I developed Falx [13], a visualization-by-example tool that can automatically synthesize programs to both prepare and visualize data from user demonstrations (Figure 1.2). Falx features a compositional synthesis algorithm that breaks down the overall synthesis process across two languages (i.e., the data transformation language and the visualization language) into small tasks that it can efficiently reason about. Furthermore, to make Falx usable, Falx is equipped with an exploration interface that lets users efficiently explore synthesized programs in the visualization space, overcoming the trust issue that users often have when interacting with previous synthesis tools [5]. Our study of Falx use by 33 data scientists shows that users can effectively and confidently adopt Falx to create visualizations that they otherwise could not implement due to their lack of programming expertise.
- Data scientists need more than programming skills to create visualizations that can faithfully deliver their insights: they also need design knowledge to properly configure their visualizations (e.g., axes type, graphical mark type). To reduce the barriers to creating good designs, I developed Draco [6], a design recommendation engine that recommends effective designs from users' partial visualization specifications (Figure 1.3). Draco formulates design principles as logic constraints and utilizes a logic

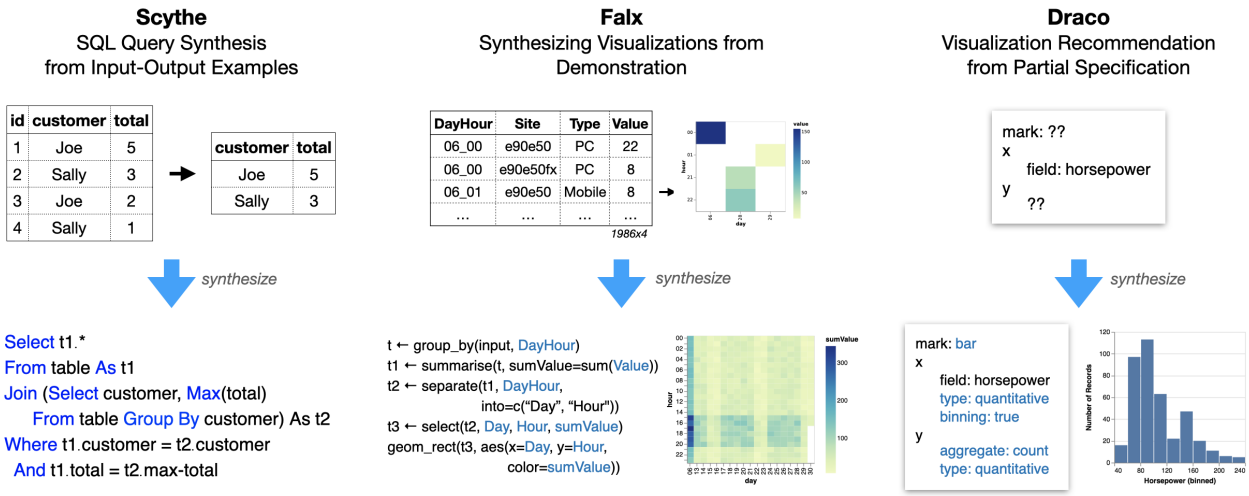

Figure 1: (1) Scythe: SQL query synthesizer from input-output examples, (2) Falx: synthesizing visualizations from demonstrations, and (3) Draco: visualization design recommendation from partial visualizations.

deduction engine to synthesize designs that maximize their effectiveness scores. It was the first extensible visualization recommendation system, and it can recommend designs from a much larger design space (2.5× larger with the same time limit of 1 second) compared to prior state-of-the art systems due to its logic formulation.

Below is a detailed view of my work and how it leads to my high-level research goal.

1 Synthesizing SQL Queries from Input-Output Examples

SQL is the de facto language for querying relational databases, and data scientists often need to use advanced operators (e.g., aggregation, outer-join, subqueries) to solve complex data manipulation tasks (e.g., computing argmax, removing duplicates). Though expressive, these operators are challenging to use, as evidenced by over 10,000 posts on Stack Overflow about these features. From these posts, I made the key observation that Stack Overflow users could often concisely describe their tasks using small input-output examples. This motivated me to develop Scythe, a SQL synthesizer that can synthesize SQL queries from small input-output examples to manipulate relational data, as forum experts can do.

The key barrier of building a practical SQL synthesizer is scalability. Prior approaches that rely on decomposability of operators to scale up are not applicable to SQL (since many SQL operators are irreversible); search-based algorithms alone are prohibitively expensive to use due to the particularly large search space imposed by SQL and the high overhead of memoizing SQL query outputs. To address this challenge, our key insight is to develop an abstraction of SQL to grant Scythe the power to reason about partial queries (queries with unfilled parameters) to allow aggressive early pruning. Concretely, given a partial query, the abstraction allows Scythe to propagate input data through it to derive an over approximation output to summarize all possible outputs that could come from queries instantiated from it. Scythe then checks the consistency between the over-approximation output and the user's output example to decide whether the partial query could lead to a correct solution and prune the whole subspace if not. This design lets Scythe dramatically prune the search space (with an average reduction of 2145× in our evaluation) with little overhead.

When tested on 193 tasks users posted in Stack Overflow, Scythe successfully solved 143 tasks (51 more than the prior state-of-the-art approach solved) and achieved an average speedup of 52× (7× to 200×) on tasks that both algorithms can solve. In fact, compared to the typical expert response time of 5 to 20 minutes on Stack Overflow, Scythe's ability to solve most tasks in 1 minute shows its potential to increase end-user productivity.

2 Synthesis-Powered Visualization Authoring

While both data transformation tools and visualization tools have been designed to reduce visualization efforts, creating expressive visualizations remains challenging: data transformation and visualization specification are two closely related tasks in exploratory analysis, requiring users to be experienced with both types of tools and iterate between them throughout the analysis process. This motivated me to develop Falx, a visualization synthesizer that can synthesize data transformation and visualization in one pass to reduce the visualization barrier.

To design an input specification that is both expressive and easy-to-use, I conducted a formative study asking participants to illustrate visualization tasks on paper. The observation that participants could easily describe visualization ideas using partial sketches motivated me to design a new programming-by-example specification, which asked users to provide partial sketches in the form of example mappings from a few input points to visual channels (i.e., graphical mark attributes like x, y-positions and color) to demonstrate their tasks. This specification generalizes the expressiveness of modern visualization grammars (mappings from columns to visual channels), allowing users to specify visualizations regardless of whether the input data is in the appropriate layout that matches the design.

From the algorithmic perspective, Falx faces two new synthesis challenges: (1) it needs to synthesize programs from two languages (visualization and data transformation languages) as opposed to one, and (2) it needs to synthesize programs from a weaker specification that consists of inputs and partial outputs as opposed to input and full output pairs. To solve the first problem, I decomposed the synthesis task into a visualization decompilation task and a data transformation synthesis task with the design of an inverse semantics for the visualization language; doing so lets Falx backwardly infer visualization programs directly from user demonstrations and breaks down the synthesis complexity. Then, to speed up data transformation synthesis, I took inspiration from bidirectional program analysis and devised a bidirectional analysis approach for Falx to better reason about partial programs. Besides propagating values forwardly from input to over-approximate outputs of partial programs (as Scythe did), it also propagates output values backwardly to compute an under-approximation of the input. This approach lets Falx derive these stronger constraints to discovery pruning opportunities that previous algorithms cannot. In evaluation, Falx solved 70 of 84 real-world visualization tasks within 20 seconds, while unidirectional abstraction only solved 49 out of 84 tasks.

Finally, to enable practical use, Falx needs to let users efficiently distinguish the correct program from other solutions that satisfy user demonstrations but do not generalize it correctly. This is known as the "disambiguation problem", which limits practical use of many program synthesis tools [5]. Instead of asking users to read and disambiguate synthesized programs, Falx transforms the challenging program disambiguation problem into a visualization exploration problem with its design of an exploration interface that lets users navigate solutions in the visualization space. Using the exploration interface, users can coarsely scan all designs to quickly rule out visualizations with high-level errors (e.g., wrong axes or labels) and then side-by-side compare similar ones in detail to choose the desired solution. By combining an efficient synthesis algorithm and a novel interface design, in our user study with 33 participants, Falx users showed statistically significant efficiency improvements on two challenging visualization tasks compared to users of R. Participants found that Falx was "easy to learn," "fast" and "can generate something that you cannot easily do otherwise," and they were "confident about solutions."

3 Visualization Design Recommendation

Falx allows data scientists to implement visualizations easily. In addition, users need design knowledge to make visualizations convey data insights faithfully and effectively. For example, the choice of many low-level visualization parameters like axis type (categorical, temporal or quantitative), starting a scale from zero or not, and using binning on an axis or not all matter, and improper designs can lead to confusing or even misleading presentations [7]. To allow non-experts with limited design knowledge to benefit from design principles that visualization research produced, I collaborated with my colleagues in visualization research and built Draco, a visualization recommendation engine that automatically suggests principally designed visualizations from partial specifications.

The key challenge in building Draco was how to model design knowledge in *a formal way* to enable

automatic reasoning and in *an extensible way* to allow Draco to adapt to new designs principles from visualization research. Drawing inspiration from formal methods research, our key insight was to model design knowledge as logic constraints to achieve both formality and extensibility. With this design, visualization experts can specify design principles in logical constraints consisting of (1) hard integrity and expressiveness constraints to prevent unfaithful designs, and (2) soft preference constraints to trade-off between different design guidelines. For end users, Draco compiles users' partial specifications into logical facts and uses a Max-SAT solver to infer designs that both satisfy hard constraints and maximize soft constraints to generate recommendations.

For evaluation, we used Draco to rebuild previous design recommendation systems that were implemented in low-level programming languages. Results showed that Draco allowed concise design principle specification (from 4,323 lines of code to 180 lines of logic rules), and it was more scalable (it explored 2.5× more search space with the same 1 second timeout). To test extensibility, we extended Draco to support task-driven recommendation from the base system. We implemented 20 new rules in a few hours and then trained Draco using data from an empirical study (1,100 pairs of visualizations rated by designers) to achieve 97% test accuracy for a design recommendation task. In addition to its recommendation power, Draco was also used as a platform to study interactions between design rules and implications of empirical studies.

4 Other Research

My HCI colleagues and I also applied program synthesis to solve the mobile layout design challenge. After observing that designers lack tools to efficiently explore the design space to prototype diverse layouts (which is essential for creating high-quality designs), we created a design exploration tool, Scout [9], which lets users explore mobile layout designs using high-level constraints based on design concepts (e.g., semantic structure, emphasis, order). Compared to prior constraint-based layout systems, which use low-level spatial constraints and generally produce a single design, Scout can efficiently synthesize a large set of principally designed layouts for exploration. In an evaluation with 18 interface designers, we found that Scout helps designers: (1) create more spatially diverse mobile interface layouts with similar quality to those created with a baseline tool, and (2) avoid a linear design process and quickly ideate layouts they do not believe they would have thought of on their own.

In addition to synthesis, I also broadly apply programming reasoning techniques to solve verification and testing problems. For example, I built a symbolic engine for SQL query equivalence checking – a known undecidable problem with many applications (e.g., verification of optimization rule correctness, superoptimization). Our solution, Cosette [2], leverages recent advances in both automated constraint solving and interactive theorem proving; it returns a counterexample (in terms of input relations) if two queries are not equivalent or a proof of equivalence otherwise. The key to scaling up Cosette is the integration of provenance analysis into constraint encoding to dramatically reduce symmetry in the search space, allowing efficient solving [12]. With these designs, despite general undecidability, Cosette can determine the equivalences of a wide range of queries that arise in practice that no previous approach can tackle, including conjunctive queries, correlated queries, queries with outer joins, and queries with aggregates. In evaluation, Cosette proved the validity of magic set rewrites and confirmed various real-world query rewrite errors, including the famous COUNT bug.

I also applied symbolic reasoning techniques for neural network safety testing and verification [10]. Motivated by the need to improve our understanding of the failures of commonly used Sequence-to-Sequence and Image-to-Sequence models, we studied the novel output-size modulation problem. First, to evaluate model robustness, we developed an easy-to-compute differentiable proxy objective that could be used with gradient-based algorithms to find output-lengthening inputs. Second and more importantly, we developed a verification approach based on convex over-approximation of neural operators that could formally verify whether a network always produces outputs within a certain length. Experimental results on Machine Translation and Image Captioning showed that our output-lengthening approach can produce outputs that are 50 times longer than the input, while our verification approach can, given a model and input domain, prove that output length is below a certain size.

5 What's Next?

In the future, I plan to develop the next generation of synthesis-powered tools to empower data scientists to solve even more challenging and more influential problems. One example task is authoring of interactive data reports. Because interactive reports can cohesively present relations of large datasets and provide readers with on-demand access to metrics of interest, they offer data insights more effectively than static reports. Due to their innate complexities, especially the need to manage correlated datasets and specify interactions, even new programming tools (like D3 and R Shiny) leave customized interactive reports only within reach of professional programmers. As interactive reports gaining popularity amongst data scientists who are not professional programmers (e.g., journalists for digital news, business people for monitoring sales trends, epidemiologists for analysis of pandemic trends, and scientists for interactive publication [8]), traditional ways to create interactive reports are no longer sufficient. If we can develop a synthesizer for interactive reports, we can democratize interactive reports and broaden the impact of data science.

These opportunities challenge program synthesis research. In particular, existing synthesizers lack (1) the scalability to synthesize programs over large languages and inputs, and (2) the expressiveness to let users concisely specify complex designs. For example, since interactive report authoring processes often involve processing large datasets using complex operators (e.g., analytical functions like window-aggregation), both the search space and the cost of deductive reasoning would grow exponentially, and existing synthesizers designed for standard operators and small input-output would not scale up. Also, since interactive reports often involve complex designs like multi-view visualizations and interactive widgets, existing interfaces like input-output examples or natural languages alone are insufficient to capture user intent. I plan to develop the next generation of program synthesizers to address these challenges from the following three perspectives:

- **Abstraction Learning.** Good language abstractions are key to scaling up synthesis algorithms since they let synthesizers reason about realizability of user specifications in a sub-search-space to find opportunities for dramatic pruning [12]. With the need to design robust abstractions for new operators or abstractions that support compositional reasoning across languages for challenging synthesis tasks, current practices of manual abstraction design no longer suffice. Instead, I plan to investigate abstraction learning approaches to allow automatic discoveries of optimal abstractions that adapt to different scenarios. Concretely, I plan to build *a reinforcement learning based abstraction learning framework* with the following components: (1) a symbolic compiler that can enumerate and propose sound abstractions for given operators, (2) a sampling-based evaluator that assesses quality of abstractions from sampled synthesis tasks, and (3) a continuous optimizer that optimizes and guides the discovery of new abstractions. In fact, our recent work showed that we could approximate language abstractions using continuous functions [1, 4], which indicated the potential to solve the abstraction search problem with continuous optimization.
- **Multimodal Program Synthesis.** In addition to better search algorithms, allowing users to provide richer task information would also help synthesizers better break the search space down to improve scalability. As the first step, I plan to design an interface that lets users demonstrate computation processes (e.g., using example formulas) besides concrete values to solve the analytical query synthesis problem, a key ingredient for building an expressive interactive report synthesizer. With these computation demonstrations, the synthesis algorithm can infer subroutines and conduct anchored search space exploration around them to improve efficiency. I will also collaborate with NLP and HCI colleagues to explore hybrid interfaces (e.g., sketching, dialog systems) to let users and computers collaboratively solve challenging programming tasks.
- **Incremental Specification.** Since I aim to target more complex design synthesis problems, it is no longer ideal to ask users to specify their ideas from scratch. Inspired by observations that software developers often copy, paste and adapt ad hoc code snippets to solve *new* tasks, I plan to build synthesizers that can synthesize programs from users' demonstrations of how to adapt an existing design to their needs and use this as the tool to solve the challenge of specifying interactive designs. To achieve this goal, I will first incorporate bidirectional programming techniques [3] to propagate user demonstrations from the canvas to program parameters and infer parameter dependencies to constrain the adaptation space to explore. I will then incorporate Draco's approach to explore this adaptation space

to recommend principled designs that are consistent with user specifications. With such an interactive specification interface, synthesizers will be able to incrementally solve complex tasks with minimal specification overhead from users.

Finally, I will assemble this research into a meta program synthesis framework and a general methodology for user-synthesizer interface design to make it easy to build synthesis-powered tools for new domains. I plan to apply this meta synthesizer to empower designers and developers in other fields to solve challenging programming tasks. For example, I will help mobile application designers directly realize graphical designs in software implementations, allow 3D model creators to create reusable parametric components from 3D meshes, and enable robot agents to distill reusable logic routines from neurally trained continuous models.

References

[1] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[2] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[3] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 341–354. ACM, 2016.

[4] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 2514–2525, 2019.

[5] Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Magazine*, 30(4):65–67, 2009.

[6] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Trans. Vis. Comput. Graph.*, 25(1):438–448, 2019.

[7] Anshul Vikram Pandey, Katharina Rall, Margaret L. Satterthwaite, Oded Nov, and Enrico Bertini. How deceptive are deceptive visualizations?: An empirical analysis of common distortion techniques. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, pages 1469–1478. ACM, 2015.

[8] Jeffrey M Perkel. Data visualization tools drive interactivity and reproducibility in online publishing. *Nature*, 554(7690):133–134, 2018.

[9] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, pages 1–13. ACM, 2020.

[10] Chenglong Wang, Rudy Bunel, Krishnamurthy Dvijotham, Po-Sen Huang, Edward Grefenstette, and Pushmeet Kohli. Knowing when to stop: Evaluation and verification of conformity to output-size specifications. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 12260–12269. Computer Vision Foundation / IEEE, 2019.

[11] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466. ACM, 2017.

[12] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):157:1–157:25, 2018.

[13] Chenglong Wang, Yu Feng, Rastislav Bodík, Alvin Cheung, and Isil Dillig. Visualization by example. *Proceedings of the ACM on Programming Languages*, 4(POPL):49:1–49:28, 2020.