Research Statement

Anders Miltner

My core goal as a programming languages researcher is to make programming more productive and programs more reliable through a combination of language design, program analysis, logic, and theorem proving. In particular, I want to enable programmers to focus on high-level design, integration, and specification, rather than low level implementation details. Over the past several years, I have made progress towards this goal through developing new techniques in formal program synthesis.

Formal program synthesis is the process of generating programs from a formal specification. These generated programs are guaranteed to satisfy the provided specification. Formal synthesis realizes the safety guarantees of traditional formal verification in a user-friendly model. By moving formal methods to design-time, programmers can focus on the high-level decisions – like "What should this piece of code do?" and "How should this code integrate into the broader software system." Formal program synthesis tools can then generate programs that are assured to satisfy the user-provided specifications. This paradigm ensures that the specifications are not written "to the code" and saves the programmer the tedious job of implementation.

These types of formal synthesizers have already found real-world applications, particularly in end-user interactions. For example, FlashFill [Gulwani, 2011] is a formal synthesizer integrated into Microsoft's Excel – it transforms input-output specifications into Excel formulas. These formulas are then guaranteed to have behavior consistent with the provided input-output examples. Because tasks like this are so simple, they permit highly efficient synthesis algorithms. But we should not focus only on simple tasks.

But how can we synthesize complex programs? To make this problem tractable, we need more information. Fortunately, the additional cost to the user of supplying that information is recovered, thanks to the additional work done by our synthesizers. The question then becomes, what information should be supplied, and what algorithms can exploit this information. With this approach, what new domains can we address? My work aims to tackle these questions.

Data Transformations

Programmers oftentimes need to perform complex transformations on data. This can happen when translating between data formats, when merging and integrating multiple data sources, when cleaning ad-hoc data, and more. These transformations are often unpleasant to write – requiring a tediously close attention to details and involving fiddly combinators.

However, as the size and complexity of data transformations increase, prior works that synthesize from simple input-output specifications begin to fall short. When generating programs from simple input-output specifications, the synthesizer must perform 3 tasks: (1) it must learn the shape of inputs (2) it must learn the shape of outputs, and (3) it must learn the function that transforms inputs to outputs. Our group made a key insight – the overall synthesis algorithm becomes much more efficient if the programmer addresses two of those tasks. Prior works tried to do everything, but it is not too difficult for programmers to describe the shape of the data, and it provides the synthesizer with a wealth of information to exploit. Our tool, Optician [Miltner et al., 2018], requires users to provide a formal description of the domain and range of the desired transformation. Optician then leverages these descriptions to generate the programs. We developed a benchmark suite consisting of transformations between ad-hoc and structured file formats coming from RedHat's Augeas tool. In our experiments, Optician synthesized all 39 of these transformations in under 30 seconds, while no prior work could synthesize more than 10 of them within a timeout window of 10 minutes.

Even after having users describe the shape of the input and output, there remains a large space of functions the system must search through to generate the desired program. To facilitate this search, we identified a key principle that helps quickly hone in on the desired transformation. Namely, users are often interested in transformations that *retain data*. The better transformation is not always the shorter one, but rather the transformation that is "closer to being bijective." In follow-up work [Miltner et al., 2019b] we formalize this intuition via information theory; our synthesized programs are those that minimize the entropy of data removed in the transformation.

Data Structures

Data structures are another key part of large software systems, and are hard to get right, even for experts. Data structures provide mechanisms for both data persistence and data abstraction. But these benefits are exactly what makes data structures tricky to verify and synthesize.

The most difficult part of data structure verification lies in invariant generation. To verify many properties of data structures, one needs to be aware of what invariants such data structures maintain throughout computation. To ensure that a BST behaves like a set, it is not enough to know that the underlying representation is a tree. One must also know that the values of the left and right descendants of each node are smaller and larger (respectively) than the node's value. We wish to automatically infer these invariants.

Prior work [Malik et al., 2008] does so in a manner analogous to fuzzing – it simply generates many random data structures, then searches for the strongest predicate ensured by all these structures. My work, Hanoi [Miltner et al., 2020], uses a more principled approach. The key is that invariant generation should be integrated with verification – we can use the properties that we aim to verify as part of the invariant generation process. We use these properties to classify candidate data structures as *valid* or *invalid*. We then use these classified structures to generate a representation invariant that holds on valid structures but not on invalid ones. If the verification fails with such a predicate, we use that failure to reclassify the structures.

But verification is only one piece of the puzzle. Ultimately, verification can identify when a programmer has messed up their data structure, but it does not help actually make correct data structures themselves. Towards to goal of synthesizing data structures, we developed Burst [Miltner et al., 2021], a bottom-up recursive synthesizer that can generate programs based on logical specifications. Because Burst can handle logical specifications, it can ensure the synthesized code does not violate the data structure invariants. But handling these logical specifications is tricky in the presence of recursion – it is unclear what recursive calls should return. To address this, Burst uses a novel notion of *Angelic Recursion*, where recursive calls can only return values consistent with the specification. When combined with a backtracking search, our algorithm is guaranteed to return programs that satisfy the provided specification.

Additionally, we believe we can integrate Burst with prior work on relational synthesis [Wang et al., 2018] to generate all the functions of a data structure at once, simply from a logical specification describing how the data structure functions should interact with each other.

Refactoring

My work on data transformations and data structures helps build new code, but a large portion of software development lies in maintenance – one of the most painful parts of which is refactoring. To alleviate this pain, a number of input-output based tools have been developed to automate the process of refactoring. These tools take a set of examples describing the code before and after the transformation as input and the synthesis engine will output a piece of code that, when run, will automatically apply such a transformation everywhere in the codebase.

However, while these tools are quite good at correctly identify the desired refactoring, they are rarely used. The amount of time and effort necessary for programmers to recognize were making a repetitive edit, find the "learn refactoring" tool, and provide input examples made these tools impractical – it is often just easier to manually edit their code! Furthermore, programmers oftentimes do not actually recognize they are making a repetitive edit. It is only when they are halfway through changing their code that developers realize they are performing a repetitive task that could be automated [Murphy-Hill et al., 2009].

Our tool, Blue-Pencil [Miltner et al., 2019a], changes how the programmer and synthesizer interact. Our key insight was that we do not need users to give explicit input-output examples, it is possible to infer a programmer's intent by watching them code. When programmers refactor their code, Blue-Pencil saves a list of snapshots of the code as it is being edited. BLUE-PENCIL then clusters similar edits, which it generates a refactoring script from.

After BLUE-PENCIL generates a candidate refactoring, BLUE-PENCIL will then nonintrusively ask the programmer whether it wants to perform the repetitive edit elsewhere in the codebase. Because of BLUE-PENCIL's general applicability and ease of use, it was included in Visual Studio 2019 as the Intellicode Suggestions feature [Groenewegen, 2020].

Future Research Goals

I am broadly interested in continuing to apply formal program synthesis to make developers' lives easier. Three concrete problems I wish to tackle are those of simplifying specifications, boosting the scalability of synthesis, and exploring novel synthesis applications.

Simplifying Specifications Complex specifications permit generating complex programs, but that does not mean there is no way to permit more simple types of specifications. I am interested in simplifying specifications while not losing synthesis capabilities.

One approach to this is multimodal synthesis. For many tasks, it is impossible to generate an intuitive specification using a single specification modality. For example, Myth [Osera and Zdancewic, 2015], a state of the art example-based synthesizer, requires 20 examples to correctly synthesize the function for inserting an element into a binary tree. Logical specifications have their own failings. The best would be some combination of the two – provide a few input-output examples, but also require that the output tree satisfies a logical specification, like the BST invariant.

Another approach is enabling natural language-based specifications. This could be done in a number of ways. One could be translation – programmers could give a natural-language specification which is translated to a formal specification. Another approach is through introducing bias. By providing an intuitive natural language description in addition to a formal partial specification, the search could be biased towards programs likely to satisfy the natural language description while ensuring satisfaction of the formal specification.

Boosting the Scalability of Synthesis While I believe neural models can help make specification easier, I find their applications to scalability issues more promising. Most programming language techniques for synthesis rely on two key approaches – deduction and enumeration. These techniques deduce portions of the program by inspecting the specification, and enumerate candidate programs via an exponential search when deduction fails. Alternatively, neural approaches like Codex [Chen et al., 2021] do not have such limitations – neural synthesizers are linear in terms of program length. Unfortunately, these neural approaches drop all correctness requirements – the generated program is not guaranteed to satisfy the specification.

I am interested in combining the strengths of each. One does not need to remove verifiable correctness to use neural models. In simplest terms, the neural model can traverse the search space, and the deduction and verification engines can define the search space, and accept or reject candidate programs.

Exploring Novel Applications With the advent of efficient algorithms, synthesis can be used to address novel problems outside the domain of software engineering. I am interested in exploring a number of such applications, though I am particularly interested in the applications of synthesis to education.

Providing new students with fully worked-out examples helps reduce the cognitive load of those learning programming [Skudder and Luxton-Reilly, 2014]. In the context of a university class, these examples can be provided by the instructor, and if individual student needs more help, they can attend office hours and get specialized attention. But what happens outside a university class? If a student is trying to self instruct, or they are attending a MOOC, they may not be able to get examples that address their individual confusions.

Synthesis can provide an answer to this problem. Say a student is learning functional programming, and they do not quite understand recursion. Course materials may have provided a few examples in terms of lists, and trees, but the student is not able to generalize to other data structures. If the student does not have access to office hours, they could simply ask a synthesizer to provide them additional examples. In addition to possible programs, these synthesizers could provide explanations for why the program satisfies the specification and how the program operates.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, et al. Evaluating large language models trained on code, 2021.

Peter Groenewegen. Making repeated edits easier with intellicode suggestions, 2020. URL `https://devblogs.microsoft.com/visualstudio/making-repeated-edits-easier-with-intellicode-suggestions/`.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011. doi: 10.1145/1926385.1926423. URL `https://doi.org/10.1145/1926385.1926423`.

Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. Deryaft: A tool for generating representation invariants of structurally complex data. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 859–862, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368223. URL `https://doi.org/10.1145/1368088.1368223`.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proc. ACM Program. Lang.*, 2(POPL), January 2018. doi: 10.1145/3158089. URL `https://doi.org/10.1145/3158089`.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019a. doi: 10.1145/3360569. URL `https://doi.org/10.1145/3360569`.

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019b. doi: 10.1145/3341699. URL `https://doi.org/10.1145/3341699`.

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385967. URL `https://doi.org/10.1145/3385412.3385967`.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up synthesis of recursive functional programs using angelic execution, 2021.

Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 287–297, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781424434534. doi: 10.1109/ICSE.2009.5070529. URL `https://doi.org/10.1109/ICSE.2009.5070529`.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015. doi: 10.1145/2737924.2738007. URL `https://doi.org/10.1145/2737924.2738007`.

Ben Skudder and Andrew Luxton-Reilly. Worked examples in computer science. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, page 59–64, AUS, 2014. Australian Computer Society, Inc. ISBN 9781921770319.

Yuepeng Wang, Xinyu Wang, and Isil Dillig. Relational program synthesis. *Proc. ACM Program. Lang.*, 2 (OOPSLA), October 2018. doi: 10.1145/3276525. URL `https://doi.org/10.1145/3276525`.