# Research Statement

Alex (Oleksandr) Polozov

December 2016

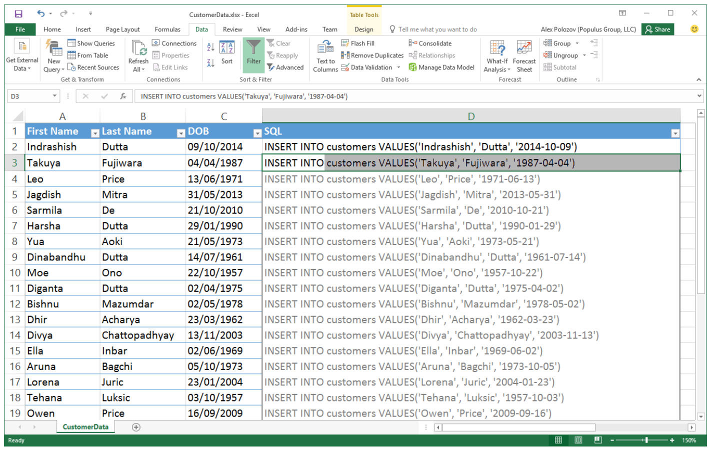
My research lies in the **automatic generation of programs** from ambiguous example-based specifications, which equips AI technologies with **domain-specific logical reasoning**, and facilitates their **industrial deployment**.

Modern AI/ML techniques are rapidly progressing to ever more complex human-like tasks. However, as the task complexity grows, so does the challenge of *practical deployment* of AI technologies. To empower arbitrary user-facing applications, AI techniques must **(a)** handle complex task domains involving multi-step actions and high-dimensional data types; **(b)** transfer and generalize learned knowledge across domains; **(c)** produce human-interpretable representations for ease of debugging and development; **(d)** be accessible off-the-shelf to a broad audience of software developers without extensive AI education. Addressing these challenges is currently an active research area in deep learning, the forefront of modern AI.

Humans deal with complex task domains by writing *programs*. Programs, in contrast to statistical models, are *universal interpretable logical representations* of actions, generic or domain-specific. The field of *program synthesis* addresses the problem of learning a program in the underlying language that is consistent with the given specification [8]. Program synthesis has been successfully applied to numerous domains including software engineering [4], robotics [3], education [18], and biology [12]. But until recently, large-scale industrial deployment of synthesizers required substantial effort.

The latest research in program synthesis shows that scaling it to real-life applications requires combining a state-of-the-art *general search algorithm* [1, 13] with a well-crafted *domain-specific language* (DSL) and some domain-specific search insight [9]. The most successful application of this approach is FlashFill, automatic synthesizer of string transformations from input-output examples in Microsoft Excel 2013 [7] (see Figure 1). FlashFill and similar mass-market example-based synthesis technologies address an important problem of *data wrangling*: preparing raw datasets into a form amenable to analysis and ML. By various accounts, data analysts spend up to 80% of their time cleaning up information instead of learning actionable insights from it [11]. This ubiquity of the wrangling problem and ease-of-use of example-based synthesis technologies led to their wide success among the users. Unfortunately, these applications were hard to develop and deploy: to achieve the required 1 sec response time, the code intertwined domain-specific logic with advanced search algorithms, making it largely unmaintainable.

To alleviate this, I developed PROSE [15, 16, 17], a general framework for automatic generation of production-quality synthesi-



INSERT INTO customers VALUES('Takuya', 'Fujiwara', '1987-04-04')

| | A | B | C | D |
|---|---|---|---|---|
| 1 | First Name | Last Name | DOB | SQL |
| 2 | Indrashish | Dutta | 09/10/2014 | INSERT INTO customers VALUES('Indrashish', 'Dutta', '2014-10-09') |
| 3 | Takuya | Fujiwara | 04/04/1987 | INSERT INTO customers VALUES('Takuya', 'Fujiwara', '1987-04-04') |
| 4 | Leo | Price | 13/06/1971 | INSERT INTO customers VALUES('Leo', 'Price', '1971-06-13') |
| 5 | Jagdish | Mitra | 31/05/2013 | INSERT INTO customers VALUES('Jagdish', 'Mitra', '2013-05-31') |
| 6 | Sarmila | De | 21/10/2010 | INSERT INTO customers VALUES('Sarmila', 'De', '2010-10-21') |
| 7 | Harsha | Dutta | 29/01/1990 | INSERT INTO customers VALUES('Harsha', 'Dutta', '1990-01-29') |
| 8 | Yua | Aoki | 21/05/1973 | INSERT INTO customers VALUES('Yua', 'Aoki', '1973-05-21') |
| 9 | Dinabandhu | Dutta | 14/07/1961 | INSERT INTO customers VALUES('Dinabandhu', 'Dutta', '1961-07-14') |
| 10 | Moe | Ono | 22/10/1957 | INSERT INTO customers VALUES('Moe', 'Ono', '1957-10-22') |
| 11 | Diganta | Dutta | 02/04/1975 | INSERT INTO customers VALUES('Diganta', 'Dutta', '1975-04-02') |
| 12 | Bishnu | Mazumdar | 02/05/1978 | INSERT INTO customers VALUES('Bishnu', 'Mazumdar', '1978-05-02') |
| 13 | Dhir | Acharya | 23/03/1962 | INSERT INTO customers VALUES('Dhir', 'Acharya', '1962-03-23') |
| 14 | Divya | Chattopadhyay | 13/11/2003 | INSERT INTO customers VALUES('Divya', 'Chattopadhyay', '2003-11-13') |
| 15 | Ella | Inbar | 02/06/1969 | INSERT INTO customers VALUES('Ella', 'Inbar', '1969-06-02') |
| 16 | Aruna | Bagchi | 05/10/1973 | INSERT INTO customers VALUES('Aruna', 'Bagchi', '1973-10-05') |
| 17 | Lorena | Juric | 23/01/2004 | INSERT INTO customers VALUES('Lorena', 'Juric', '2004-01-23') |
| 18 | Tehana | Luksic | 03/10/1957 | INSERT INTO customers VALUES('Tehana', 'Luksic', '1957-10-03') |
| 19 | Owen | Price | 16/09/2009 | INSERT INTO customers VALUES('Owen', 'Price', '2009-09-16') |

**Figure 1:** FlashFill in Microsoft Excel, automatically synthesizing a string transformation macro from a single input-output example [7]. The learned macro involves conditional statements, regex matching, and date processing logic.

]

zers from minimal domain definitions. PROSE takes as input a DSL with some expert knowledge and combines it with a novel search algorithm to produce an efficient synthesizer, tailored to a particular domain. Over the past two years, engineers and researchers (both within and outside Microsoft) have built 10+ diverse tools on top of PROSE, with applications including data cleaning [6, 14, 20], software refactoring [21], and education [10]. Many have been deployed in mass-market Microsoft products such as Cortana Intelligence Suite, Exchange, PowerShell, and Azure Log Analytics.

Effectively leveraging domain-specific insight and abstraction turned out imperative to scaling up program synthesis. I believe that statistical ML techniques currently require a similar development to progress to the next level of AI capabilities. Building upon my previous work, my immediate goal is to augment statistical ML with logical inference and abstractions, successfully applied before in program synthesis, and then use these instruments to derive and re-use new domain definitions.

## 1 Program Synthesis using Examples

Program synthesis is an incredibly complex combinatorial search problem. Given user intent in the form of some specification (*e.g.*, input-output examples, natural language descriptions, Boolean assertions), it searches an underlying programming language for a program consistent with this intent. State-of-the-art approaches tackle this complexity using *syntax-guided synthesis (SyGuS)*: they restrict the target program's syntax to a given grammar or a partial sketch [1, 24]. In this methodology, every synthesis application is a specialization of a generic search algorithm with a DSL.

Even with modern developments in search algorithms and SMT solving, applying program synthesis to real-life underspecified domains like string transformations has proved challenging. For this reason, industrially deployed synthesizers such as FlashFill included a fair amount of domain-specific logic for search space reduction. Unfortunately, it also meant that domain-specific knowledge and generic search were fundamentally intertwined in the codebase. Deployment and maintenance of such software is onerous. Here is an insider example: FlashFill took *two person-years* of PhD-level work to build, and then an additional team of engineers to integrate it into Excel. To continue deploying synthesizers in the wild, we had to reduce this complexity.

### 1.1 PROSE Framework

The key insight in scaling up program synthesis was to separate it into domain-specific and domain-agnostic components. I realized that 10+ previously developed synthesis applications could be generalized using one search strategy that explores the grammar of the corresponding DSL in a particular way. The search is guided by tiny snippets of domain knowledge, which we call *witness functions*. These functions approximate inverse semantics of the DSL operators, which allows PROSE to *backpropagate* the provided input-output examples through the grammar top-down, deductively inferring the subexpressions of the desired program.

Backpropagation is effective in PROSE for two reasons. First, deductive inference achieves real-time synthesis performance (*i.e.,* in 1-5 seconds, as opposed to minutes/hours with an SMT solver), which is essential to a user-facing application. Second, it makes program synthesis *modular*: all domain-specific insight is separated into witness functions, which do not reference any knowledge about the entire synthesis algorithm. They are written once and re-used in different domains. Moreover, they can be written by domain experts who are not proficient in formal logic, AI, or program synthesis. In the field of program synthesis, this is the first implementation of a general-purpose search algorithm that can be guided by third-party domain-specific insight.

As a result, PROSE *speeds up synthesizer development from 2 years to 2 months*, all with a production-quality artifact on the output [16, 17]. The produced applications perform example-based synthesis in complex real-life domains such as regular expressions, Web mining, and databases.

### Real-life example

In our latest work, we applied PROSE to learning *repetitive code transformations* [21]. Repetitive code editing commonly occurs in **(a)** *software refactoring*, where developers edit code in a way that likely occurred in a different part of their codebase, and **(b)** *programming education*, where students edit their solutions in a way that likely occurred in other students' solutions. In both scenarios, our tool, REFAZER, uses these repetitions as input-output examples for learning abstract code transformations.

We successfully used REFAZER to refactor large open-source C# projects on GitHub (Roslyn, NuGet, Entity Framework) and to automatically construct feedback for 720 students in UC Berkeley's CS61A *massive open online course* [10]. In the latter case, it immensely reduced the workload of the course instructors. They were able to quickly explore *clusters* of buggy submissions with the same fault, apply the learned transformation to fix them, and provide high-level feedback to all affected students.

I want to point out the benefits of PROSE for this project: (a) REFAZER was developed in a few months and immediately used to aid the course TAs, (b) PROSE facilitates concept reuse, which made REFAZER general enough to be used for two widely different applications of code transformations.

### Microsoft PROSE Team

Creation of PROSE allowed my advisor, Sumit Gulwani, to establish a full-time Microsoft R&D team of 10+ researchers and engineers who develop and deploy a library of example-based synthesis technologies on top of it. As a founding member, I have worked closely with the group over the past two years, leading research and development of the PROSE core framework.

Our partner teams have deployed PROSE-based applications in PowerShell (cmdlets for string processing), Cortana (knowledge acquisition from emails), Azure Log Analytics (custom field creation in logs), and other Microsoft products. Before the framework, such agility would be impossible: FlashFill required 2+ person-years for its one-of-a-kind deployment. We also publicly released PROSE for academia, which, among other engagements, enabled UC Berkeley researchers to create REFAZER.

## 1.2 Intent Disambiguation

The primary challenge of example-based program synthesis is *intent ambiguity*. In contrast to ML, a synthesis problem is defined by only a handful of samples. Thus, every problem is terribly underspecified: a typical DSL may contain up to $10^{20}$ programs consistent with a given input-output example [16]. Nevertheless, superb user experience demands to guess the user's intent correctly from as few examples as possible. As we observed, with more examples the users quickly lose trust in the synthesis system and avoid re-using it.

At the time of PROSE development, the state-of-the-art approach for disambiguating user intent was *ranking*. It relies on a *ranking function* $\ell \colon P \to \mathbb{R}$ that scores each DSL program according to its “robustness” or “likelihood of being correct in general.” Designing a good ranking function is difficult and is largely an art. For example, simply preferring shorter programs is unsatisfactory because it also prioritizes short constant subexpressions that overfit the examples. As a result, manually crafted ranking functions tend to accumulate special cases and turn into a complex zoo of heuristics. Prior work has explored learning ranking functions automatically with ML [22], but this is only possible with a large dataset of completed tasks, rarely available during initial domain development.

As we started developing PROSE-based applications en masse, we found that *user interaction* was a much more productive model. Since our user is ultimately the main judge of her intent, we should rely more on her judgment in the face of uncertainty. We compared three models of interaction when the learned program may not satisfy the overall intent:

1. Accepting *negative examples*, which only point out wrong outputs without correcting them.
2. Displaying the program to the user (either directly or paraphrased in natural language).
3. Asking proactive *disambiguating questions* on the inputs where the learned programs disagree.

Out of these, the third model proved most useful [15]. Our questions make the users complete their wrangling tasks with fewer mistakes because the questions focus the users’ attention on the ambiguities in their tasks without the need to eyeball the entire dataset. Moreover, proactive help in the form of clarifying questions also boosts the users’ confidence and trust in the synthesis system. We are currently integrating such feedback-based disambiguation UX on top of PROSE into several mass-market Microsoft products.

I believe that this finding is relevant to many modern user-facing AI applications: *mixed-initiative interfaces where the users provide active feedback based on the system’s suggestions are more helpful than completely opaque automation, however perfectly implemented*.

### Limitations

Program synthesis as a form of ML is complementary to statistical techniques. Unlike ML, it handles high-dimensional data types well, and it learns the structure of its produced artifact in addition to the parameters. However, it also lacks the ability to generalize from noisy data: the user’s input-output examples are assumed to be precise. It also relies on engineered DSLs to guide the search process (akin to manually crafted features in ML). Incorporating statistical techniques into logical inference to overcome these limitations is an active research area in our field [19]. Moreover, I believe that new developments in AI *require* drawing on the strengths of both fields.

## 2 Future Vision

Over the last decade, program synthesis performed a great leap forward, exemplified in learning complex programs from loose specifications in mass-market applications. I believe that its ability to perform logical reasoning and leverage domain-specific insight will provide a new level of capabilities to modern AI technologies. Machine learning has long realized the importance of proper representations to effective learning. Nowadays, adopting *programs* as the underlying representation of AI promises to resolve the omnipresent demand to make AI artifacts debuggable and interpretable.

This vision has prompted a series of research projects: from TerpreT, a specification language for inference problems [5], to inducing programs as alternative explanations for learned ML models [23].

However, productively combining logical and statistical techniques for program synthesis remains an open research problem. Preliminary results suggest that insights from one field can simplify problem definitions for the other [2] but combining both into a single algorithm requires expertise from both sides of AI. As the first step in this mission, I want to adapt logical inference of PROSE and modern deep learning for the following immediate AI challenges.

**Operator learning** In synthesis, we use DSLs to build levels of abstraction and parameterize learning algorithms with third-party domain definitions. DSL operators and structure is a form of *domain knowledge*: they serve as a *prior* for synthesizing useful programs from reusable building blocks. State-of-the-art approaches design such DSLs manually, which often hinders said reusability.

This situation can be likened to *feature design* in ML a decade ago. Deep learning improved it with its automatic derivation of complex features from atomic components of the problem (*e.g.,* pixels in an image or words/characters in a sentence). Applying a similar technique to common synthesis domains may eliminate the manual burden of designing and ranking custom DSLs. Moreover, I believe that it will provide novel insight into properties of our DSL operators that make them efficiently synthesizable.

**Knowledge transfer** Program synthesis, like the entire field of software engineering, relies heavily on *abstraction* and *modularity*. DSLs, their operators, and underlying SMT theories serve as a form of *knowledge transfer* in program synthesis: they allow us to mix and match synthesis algorithms from different domains.

In contrast, deep learning can learn high-level features for a domain of problems, but these features fail to generalize to other domains. Performing such generalization amounts to *inducing* a rule from the learned features—a problem one level higher than the typical ML problem of inducing a feature from the samples. I believe that induction and deduction techniques that work spectacularly in constraint-based synthesis apply equally to the domains that are typically associated with ML. Preliminary work shows successful applications of such techniques to *biological inference* [12, 25].

**Human-interpretable models** The black-box nature of deep learning and its lack of debuggability is a major open problem in AI. While neural models by themselves are difficult to interpret, program synthesis and DSLs can help in this regard. There are multiple ways to apply them to the deep learning artifacts:

- **(a)** use features/subroutines that were learned by techniques from the previous paragraph as a high-level interpretation;
- **(b)** synthesize an "interpretation" program from a supplementary DSL that most closely approximates the model as a black-box function;
- **(c)** combine both approaches by inducing a supplementary DSL from the learned subroutines.

Any of these approaches makes ML-based AI more transparent, which helps to apply it to new domains on an industrial scale.

Ultimately, combining program synthesis and ML will make AI applicable to the whole new category of applications: those that require logical reasoning and domain understanding in addition to pattern recognition. It will also vastly simplify its industrial adoption, by making AI technologies available to domain experts without ML education, and providing transparency into the learning process. Achieving both of these goals will make a crucial next step to bringing AI to the same level of ubiquity as programming is today.

## References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, et al. “Syntax-guided synthesis”. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2013, pp. 1–17.

[2] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, et al. “DeepCoder: Learning to Write Programs”. In *arXiv preprint* (2016). URL: http://arxiv.org/abs/1611.01989.

[3] A. Feniello, H. Dang, and S. Birchfield. “Program synthesis by examples for object repositioning tasks”. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 4428–4435.

[4] J. Galenson, P. Reames, R. Bodik, B. Hartmann, et al. “Codehint: Dynamic and interactive synthesis of code snippets”. In *36th International Conference on Software Engineering (ICSE)*. ACM. 2014, pp. 653–663.

[5] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, et al. “TerpreT: A Probabilistic Programming Language for Program Induction”. In *arXiv preprint* (2016). URL: http://arxiv.org/abs/1608.04428.

[6] M. I. Gorinova, A. Sarkar, A. F. Blackwell, and K. Prince. “Transforming spreadsheets with data noodles”. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2016, pp. 236–237.

[7] S. Gulwani. “Automating string processing in spreadsheets using input-output examples”. In *Proceedings of the 38th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 2011.

[8] S. Gulwani. “Dimensions in program synthesis”. In *12th Symposium on Principles and Practices of Declarative Programming (PPDP)*. ACM. 2010, pp. 13–24.

[9] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, et al. “Inductive Programming Meets the Real World”. In *Communications of the ACM* 58.11 (2015), pp. 90–99.

[10] A. Head, E. Glassman, G. Soares, R. Suzuki, et al. “Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis”. In *4th Annual ACM Conference on Learning at Scale*. L@S’2017. ACM, 2017.

[11] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. “Enterprise data analysis and visualization: An interview study”. In *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926.

[12] A. S. Koksal, Y. Pu, S. Srivastava, R. Bodik, et al. “Synthesis of biological models from mutation experiments”. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*. 2013.

[13] T. A. Lau, P. Domingos, and D. S. Weld. “Version Space Algebra and its Application to Programming by Demonstration.” In *Proceedings of the 17th International Conference on Machine Learning (ICML)*. 2000.

[14] V. Le and S. Gulwani. “FlashExtract: A framework for data extraction by examples”. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*. ACM. 2014, p. 55.

[15] M. Mayer, G. Soares, M. Grechkin, V. Le, et al. “User Interaction Models for Disambiguation in Programming by Example”. In *28th ACM Symposium on User Interface Software and Technology (UIST)*. 2015, pp. 291–301.

[16] O. Polozov and S. Gulwani. “FlashMeta: A Framework for Inductive Program Synthesis”. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. 2015, pp. 107–126.

[17] O. Polozov and S. Gulwani. “Program Synthesis in the Industrial World: Inductive, Incremental, Interactive”. In *5th Workshop on Synthesis (SYNT)*. 2016.

[18] O. Polozov, E. O’Rourke, A. M. Smith, L. Zettlemoyer, et al. “Personalized mathematical word problem generation”. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[19] V. Raychev, P. Bielik, M. Vechev, and A. Krause. “Learning programs from noisy data”. In *ACM SIGPLAN Notices*. Vol. 51. ACM. 2016, pp. 761–774.

[20] M. Raza and S. Gulwani. “Automated Data Extraction using Predictive Program Synthesis”. In *31st AAAI Conference on Artificial Intelligence*. To appear. 2017.

[21] R. Rolim, G. Soares, L. D’Antoni, O. Polozov, et al. “Learning Syntactic Program Transformations from Examples”. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. To appear. 2017.

[22] R. Singh and S. Gulwani. “Predicting a correct program in programming by example”. In *27th International Conference on Computer-Aided Verification (CAV)*. 2015.

[23] S. Singh, M. T. Ribeiro, and C. Guestrin. “Programs as Black-Box Explanations”. In *Neural Abstract Machines and Program Induction Workshop at NIPS*. 2016.

[24] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[25] M. J. Sternberg, A. Tamaddoni-Nezhad, V. I. Lesk, E. Kay, et al. “Gene function hypotheses for the Campylobacter jejuni glycome generated by a logic-based approach”. In *Journal of molecular biology* 425.1 (2013), pp. 186–197.