Title
Abstraction, Generalization, and Embodiment in Neural Program Synthesis

Author
Shin, Eui Chul

Publication Date
2020

Peer reviewed|Thesis/dissertation

Abstraction, Generalization, and Embodiment in Neural Program Synthesis

by

Eui Chul Shin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor Alvin Cheung
Professor David Bamman

Summer 2020

Abstraction, Generalization, and Embodiment in Neural Program Synthesis

Abstract

Abstraction, Generalization, and Embodiment in Neural Program Synthesis

by

Eui Chul Shin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Program synthesis, or automatically writing programs from high-level specifications has been a long-standing challenge in computer science and artificial intelligence. Addressing this challenge can help unlock the full power of computing to nontechnical users, assist existing developers on traditional programming tasks, and solve other tasks in artificial intelligence like question answering that are naturally expressible as programs. In recent years, neural methods for program synthesis based on learning have driven significant progress. With this shift, themes like abstraction, generalization, and embodiment that recur in other facets of machine learning provide a natural framework for further improvement. In this dissertation, we present methods to address manifestations of these themes in several concrete instantiations of neural program synthesis.

First, we demonstrate how to better synthesize imperative programs by interacting with the program interpreter environment in the form of predicted execution traces, in a challenging domain for program synthesis called Karel. We also show in empirical studies that generating synthetic data for program synthesis requires significant care to enable models to generalize. In an application of program synthesis from natural language, or semantic parsing, we present attention-based neural architectures that can better encode the natural language specification to enable better generalization to new database domains. In this and other code generation domains, we introduce a method for integrating automatically learned code idioms into the synthesis procedure, learning to automatically switch between multiple levels of abstraction.

Contents

List of Figures

List of Tables

Acknowledgments

I am very grateful to have received so much support during my PhD and for all the work leading up to this dissertation, First, I would like to thank my advisor Dawn Song, who first got me started in research as an undergraduate student. Since that time and later during my grad school years, her guidance, support, mentorship, and encouragement have been critical for my development as a scientist and researcher. Her boundless energy, long-term vision, and technical insight has been a significant inspiration. I am particularly grateful for her support and encouragement as I explored different areas of research throughout grad school.

I am highly fortunate to have had many other mentors and senior collaborators during my time at Berkeley. In particular, I am grateful to have worked with and learned from Chia Yuan Cho, Daniel Reynaud, Devdatta Akhawe, Domagoj Babic, Emil Stefanov, Neil Gong, and Prateek Saxena when I was a young and naïve undergrad. I would also like to thank Gilad Katz, John Schulman, Kevin Chen, and Mario Frank for their patience and kindness during the early years of my PhD.

I am grateful for the help of professors Doug Tygar, Joan Bruna, and Pieter Abbeel for serving on my qualifying exam committee, and Alvin Chen, David Bamman, and Koushik Sen for participating in my thesis proposal defense and dissertation committee. Thanks for your time, feedback, and all the advice.

Outside of Berkeley, I have also greatly benefited from several internships throughout grad school. Thanks to Wolfgang Macherey, George Foster, Maxim Krikun, and others in Google Translate for their collaboration and mentorship during my first internship as a PhD student. I learned a lot from my work with Geoffrey Irving, Alex Alemi, Sarah Loos, Christian Szegedy, and Oriol Vinyals during my second internship. I would like to thank Illia Polosukhin and Alex Skidanov for taking a chance on me after a brief meeting at NeurIPS, all the brainstorming and fun discussions about program synthesis, and giving me my first experience with a start-up. I am indebted to Alex Polozov for a highly enjoyable summer in Seattle at Microsoft Research where I also had the pleasure of collaborating with Miltos Allamanis and Marc Brockschmidt. Thanks to Vladlen Koltun for hosting me as an intern outside of his usual research area and providing an environment that led to the final chapter of this dissertation.

I would like to thank my colleagues in Dawn's research group for enriching exchanges of ideas, feedback on paper drafts, camaraderie, and other help. In addition to those mentioned elsewhere in this section, I would like to thank Bo Li, Chang Liu, Chao Zhang, Dan Hendrycks, Jonathon Cai, Min Du, Mitar Milutinovic, Noah Johnson, Ruoxi Jia, Warren He, and Xinyun Chen. During my internship at Microsoft Research, I was fortunate to share an office with Austin Tran, Ishan Durugkar, and in particular with Evan Pu who was always an endless source of ideas. Thanks to Rishabh Singh, for help in framing the paper which became the basis for the second chapter of this dissertation, career advice, and other topics. It was also a pleasure to collaborate with Roy Fox, Sanjay Krishnan, Ion Stoica in the RISE Lab. I am grateful to have learned from many others in the Berkeley security group including Alex Kantchelian, Brad Miller, Chris Thompson, Nicholas Carlini, and Paul Pearce.

Special thanks to all the undergraduate and master's students I worked with during my PhD, including Eugene Che, Himadri Mishra, Joel Simonoff, Jonathan Wang, Kimberly Lu, Mark Wu,

Nipun Ramakrishnan, Seth Park, Tu Ni, William Paul, Xiaoyuan Liu, Xu Zou, Yifan Li, and Yitian Zou. In particular, I would like to thank Brandon Trabucco, Chris Bender, Kavi Gupta, and Neel Kant for the long and fruitful collaborations, including the work that formed the basis of the second chapter of this dissertation. All of you have all taught me a lot and I am grateful for your hard work, enthusiasm, and patience.

I want to acknowledge the hardworking and competent staff in the department, including Angie Abbatecola, Anthony Stamos, Barbara Goto, Jean Nguyen, Lena Lau-Stewart, and Shirley Salanio, who helped smooth out so many obstacles for me and other students.

I am also thankful for the friends I had at Berkeley, without whom my PhD would have been a much lonelier experience. Thanks to all the officemates I had over the years for all the interesting discussions and company: Erika Chin, Cynthia Sturton, Thurston Dang, David Fifield, Michael McCoyd, Charles Packer, Daniel Huang, Melih Elibol, and Rachel Lawrence. I am particularly thankful to Austin Murdock, Frank Li, Grant Ho, Nathan Malkin, Pratyush Mishra, and Rishabh Poddar for mutual support in quarantine and fun outings before. Thanks to Cathy Wu, Fanny Yang, and Philipp Moritz (and later Ludwig Schmidt and Reinhard Heckel) for all the dinners with soft tofu soup and goat cheese-free pizza. Thanks to Jethro Beekman for organizing so many board game nights (and mornings and afternoons) where I had great fun playing with Ben Keller, Charles Reiss, Nathan Narevsky, Stephen Twigg, and others. Thanks to Robert Nishihara for organizing several outings to San Francisco, including karaoke nights at which I never managed to sing.

Outside of grad school, I would like to thank Charles Chen, Christopher Lin, Darren Kuo, and Yiding Jia for their enduring friendship and video game sessions. Thanks to Alexander Mills, Arun Ganesan, Michael Huang, and Nima Khazaei for all the support over the years since the robotics club in Ann Arbor. I am also grateful for Little Mountain, a big house in Elmwood that was my home for most of grad school, and everyone who lived there for all the conversations, co-working, cooking together, and even some occasional parties. In particular, I would like to mention Atsuya Kumano, Audrey Tiew, Ben Weitz, Cathy Wu, Chi Jin, Colleen Josephson, Elise Guinee-Cooper, Emerald Ferreira-Yang, Eric Price, Frank Li, Jordan Sullivan, Julia Christiansen, Kyle Miller, Lynn Yi, Ma'ayan Bresler, Olivia Angiuli, Paul Christiano, Philipp Moritz, Rebecca Chanis, Vinay Ramasesh; and Rishi Gupta for getting the house started.

I would like to thank my parents Eonyoul Shin and Eonhee Yoo and my siblings Jisue and Edward for all their love and encouragement throughout the years, who have been foundational in letting me grow into the person I am today. And finally, thanks to Ian Gulliver for his companionship, patience, encouragement, and unwavering support, that really made a difference during the last years of grad school.

During the long journey of graduate school, I was fortunate to receive support from many wonderful people including some that I must have missed from listing here. To all—thank you!

Chapter 1

Background and Introduction

Over the past several decades, computing has reshaped our world and how we communicate, entertain, learn, work, and think. The concept of *software* has played a central role in this shift, by establishing a separation between the hardware making up the electrical and mechanical form of a machine, and the ultimate work that it carries out in the form of a sequence of logical and mathematical operations. This separation enables the same hardware to serve an endless array of uses, even those that the creators of the hardware had never envisioned. Furthermore, it has catalyzed the rapid development of products and services, as the increasingly complex structures necessary to achieve these larger goals are much easier to build up in software, free from many constraints imposed by the physical world. These products and services have become economically valuable to an enormous extent such that today, the majority of the world's top 10 most valuable companies have software as the central ingredient of their offerings.

These software offerings provide significant value to the world as a whole, and they reach more people now than ever. Nearly half of the world are internet users, and the number of mobile subscriptions has exceeded the world population (World Bank 2018a; World Bank 2018b). However, the vast majority of these people only interact with the software as end users, within the bounds prescribed by the creators of existing software. Without knowledge about how to program, they have little recourse if the existing software doesn't operate the way they want. The full power of computing remains accessible only to the small minority who have this knowledge. This small minority has been expanding, as learning how to program has become much more popular in recent years, for example as witnessed with the growth of computer science enrollment on university campuses. Nevertheless, even though the field of programming education has come a long way, for many programming remains a difficult and time-consuming skill to learn.

Even for those proficient at programming, it often proves a treacherous endeavor. Since computers operate inflexibly on their provided instructions, programmers need to pedantically specify details they might rather elide. While the development of higher-level programming languages, software libraries, and software engineering techniques has improved the situation over time, there remains much room for programmers to make mistakes that result in bugs; sometimes mere annoyances, but at other times, causing serious damage to lives (Leveson and Turner 1993), businesses (Popper 2012), and even space probes (Oberg 1999). Furthermore, software commonly

interfaces with untrusted users, for example over the internet, including adversaries motivated to find previously undiscovered bugs that can be exploited for their own gain. Beyond these concerns of bugs and security issues, ensuring that a program is not only *correct* but also *fast* and *resource-efficient*, presents an additional layer of challenges. At the extremes, such work is reserved for specialists with arcane knowledge on the underlying hardware and operating system, and can thus mold the program to fit as needed. In recent years, taking full advantage of the hardware has also meant writing concurrent programs, which is notoriously hard considering the ease of introducing issues like deadlock.

The considerations above motivates *program synthesis*: getting computers to program themselves. Provided with a high-level specification, we would like to automatically find a program that fulfills that specification. This problem has been studied since the early days of computer science, in different communities such as programming languages and artificial intelligence (Waldinger and Lee 1969; Manna and Waldinger 1971). Despite the significant progress that has been made since those early days, program synthesis remains an extremely challenging task. Contemporary methods can only synthesize programs of limited complexity, and only succeed on a small subset of high-level specifications that users might provide. This dissertation's goal is to address these gaps and bring us closer to this longstanding dream.

1.1 Program Synthesis Through Formal Methods

In the beginning, and more recently within the programming languages community, program synthesis has been approached mostly through the lens of formal methods. The choice of approach also determines which kinds of specifications we can use. In an early example, Green (1969) used automated theorem proving methods to transform carefully-specified formal specifications for problems like the Tower of Hanoi into an executable program. More generally, we can imagine approaches which start with the specification as given, and apply a sequence of transformations to it until we have the desired program. However, such approaches are impractical for the majority of cases where formally specifying the desired program's behavior at a sufficient level of detail is no easier than writing the program itself.

As such, a large body of subsequent work has focused on less formal but more tractable forms of specifying program behavior. A classic format is input-output pairs, where for a small number of inputs (sometimes just one), the user specifies the desired output from the hypothetical program. Flash Fill (Gulwani 2011) is a notable example of this paradigm, also called *programming by example*, where the system can generate regular expressions for string transformations given one example. For this kind of problem, top-down divide-and-conquer search approaches can work well. We recursively choose a top-level operator $F(x_1, \cdots, x_n)$ that should appear at the root of the program, determine what specification each x_i needs to specify considering F, and repeat for each of the x_i.

Another well-known approach is programming by sketching (Solar-Lezama 2013), wherein the user provides a partial program with holes (a *sketch*) and an executable specification/verifier for the desired program. We can then search the space of programs (filling in possible values for the holes)

until we find one that meets the specification. By expressing this search procedure in the right way, we can make use of battle-tested general-purpose tools like SAT and SMT solvers to explore the space efficiently.

For a comprehensive introduction, we refer the reader to Gulwani et al. (2017), a detailed survey on related approaches to program synthesis.

1.2 Limitations of Formal Methods

Program synthesis methods based on search and formal reasoning have enabled significant applications like automated data cleaning and extraction (through string transformations), optimizing programs, and bug-finding. Nevertheless, there are several fundamental limitations with the approach.

1.2.1 Only Formal Specifications

First, these methods only work for specifications that are amenable to formal reasoning or execution. As discussed earlier, it is uncommon that a complete and formal specification about the program's desired behavior is easier to write than the program itself, and tools relying on such specifications are accessible to an even smaller set of people than computer programmers. In contrast, input-output pairs are very intuitive and a good fit for certain applications like string manipulations, but unsuitable for other kinds of programs where computing the output is hard without having the program in the first place (for instance, matrix multiplication or finding shortest paths in graphs).

When people communicate with each other, even on matters pertaining to computers and programs, they use modalities such as natural language, drawings, and demonstrations. Formal reasoning methods provide no simple route towards interpreting these kinds of communication. We can try to translate them into formal specifications first, but the inherent ambiguity and noise makes that a challenging and ill-specified task. A more broadly useful program synthesis system needs to support these kinds of modalities.

1.2.2 Heuristics for Choosing Solutions

Even if a specification is formal, that does not mean it is unambiguous. A single input-output pair can conveniently specify a string transformation program, but typically, there are a very large number of possible programs which would satisfy that pair. Even if we solicit additional input-output examples, that may not sufficiently disambiguate among the possible programs despite the extra effort spent by the user. The system needs to make its best guess for which program the user intended based on limited information.

Doing so well requires careful work from the designer of the program synthesis system. The designer must provide various heuristics to favor or disfavor certain kinds of programs. For the string-editing case, the following example from Gulwani (2011) is illustrative of the kinds of heuristics needed:

> A `Concatenate` constructor is simpler than another one if it contains smaller number of arguments or its arguments are pairwise simpler. [...] StartTok and EndTok are simpler than all other tokens (suggesting that extraction logics based on the start/end of strings are more common). [...] `CPos` expressions are simpler than `Pos` expressions (giving preference to extraction logics based on constant offsets).

We can also design the *domain-specific language* that the system programs in, such that programs implementing the user's probable intentions are straightforward to express, but other kinds of nuisance programs that happen to satisfy the specification are difficult or impossible. For example, given an input-output example for a string transformation, the user would not have intended a lookup table mapping the provided input to the corresponding output, and returning an arbitrary output otherwise.

1.2.3 Heuristics for Searching

The design of the domain-specific language is also important because it defines the search space over possible programs. The number of unique programs of a given length grows exponentially with the length. Unless the search procedure can exclude most of the search space, long programs are very difficult to find. The difficulty of searching further increases the importance of using a carefully-designed domain-specific language. Although such a language would necessarily limit the kinds of programs we can synthesize, limiting the generality of the system, it allows us to concisely express concepts within a narrower target domain, so that we can synthesize programs that might have been exceedingly long in a more general-purpose language.

The use of solvers, like SAT or SMT solvers, can also enable us to outsource the actual searching to a specialized tool. These solvers contain extensive optimizations and clever data structures. They use many algorithms, heuristics, and data structures that work well in practice for many problems. However, since the underlying task is NP-hard, successful use of the tools can require a great deal of experience gained through trial-and-error that informs what the tools can do well and what they cannot.

1.3 Deep Learning

In the previous section, we discussed the limitations of program synthesis methods based on formal methods. Overall, these limitations share similarities to those faced by early symbolic approaches to artificial intelligence, which has limited ability to deal with uncertainty and required intractable amounts of computation to scale to larger problems. Since then, artificial intelligence research has largely shifted to machine learning, with a focus on building systems that solve problems by learning patterns from provided data and gained experience.

Over the past few years, deep learning has become the dominant paradigm within machine learning. The field has its origins in artificial neurons to model neural activity in brains (McCulloch and Pitts 1943). The perceptron (Rosenblatt 1958) introduced learning artificial neurons from

data, and backpropagation (Werbos 1974; Parker 1985; Rumelhart et al. 1986) enabled learning of multiple *layers* of artificial neurons in a *neural network*. Amongst the large amount of research in the field, recognizing handwritten digits (LeCun et al. 1989) was an early example of a practical application of neural network learning.

About two decades later on, a neural network-based method (with a similar architecture to LeCun et al. (2015), but with a larger number of parameters and trained on much more data) won the ImageNet Large Scale Visual Recognition Challenge in 2012 (Krizhevsky et al. 2017; Russakovsky et al. 2015), a competition about object recognition in images. The name "deep learning" also became popular around this time, recognizing that these and similar methods use neural networks containing many layers. Since those times, deep learning methods have become the standard for many application domains: other domains within computer vision such as object recognition, segmentation, and reconstruction; automatic speech recognition and speech synthesis; natural language processing tasks like machine translation, semantic parsing, and language modeling; robotics and control; and many others.

A compelling aspect of deep learning is creating higher-level representations directly from raw data, avoiding hand-crafted feature extractors needed by other methods. Although much of the work in building feature extractors has now been replaced with building neural architectures and optimization procedures, this shift has also enabled greater cross-pollination of common tools and ideas between disparate application domains. For a more comprehensive overview on the subject and an introduction to the technical details, we refer the reader to LeCun et al. (2015) and Goodfellow et al. (2016) respectively.

1.4 Neural Program Synthesis

This dissertation focuses on neural program synthesis: approaching program synthesis through the methods of deep learning. As mentioned in the previous section, deep learning provides tools to learn directly from data, without the need for carefully-designed feature extractors. In program synthesis, we want to transform a high-level specification for the desired program into the code that implements it. In neural program synthesis, we use neural networks to learn this mapping from specifications to programs; and to a large extent, the same neural architectures used in other domains can transfer directly to similar tasks.

1.4.1 A High-Level Framework for Neural Program Synthesis

Figure 1.1 provides an overview of the approach. The specification for the program may take several different forms, such as input-output examples in symbolic form (as shown in the table), a short natural-language description (a question against a database), or even a geometric or pictorial representation (not depicted in the figure) of the input or output if natural for the program's domain. We can *encode* this specification using common neural architectures, such as long short-term memory (Hochreiter and Schmidhuber 1997) used for sequences in fields like natural language processing, convolutional neural networks (Fukushima 1980; LeCun et al. 1989) used in computer

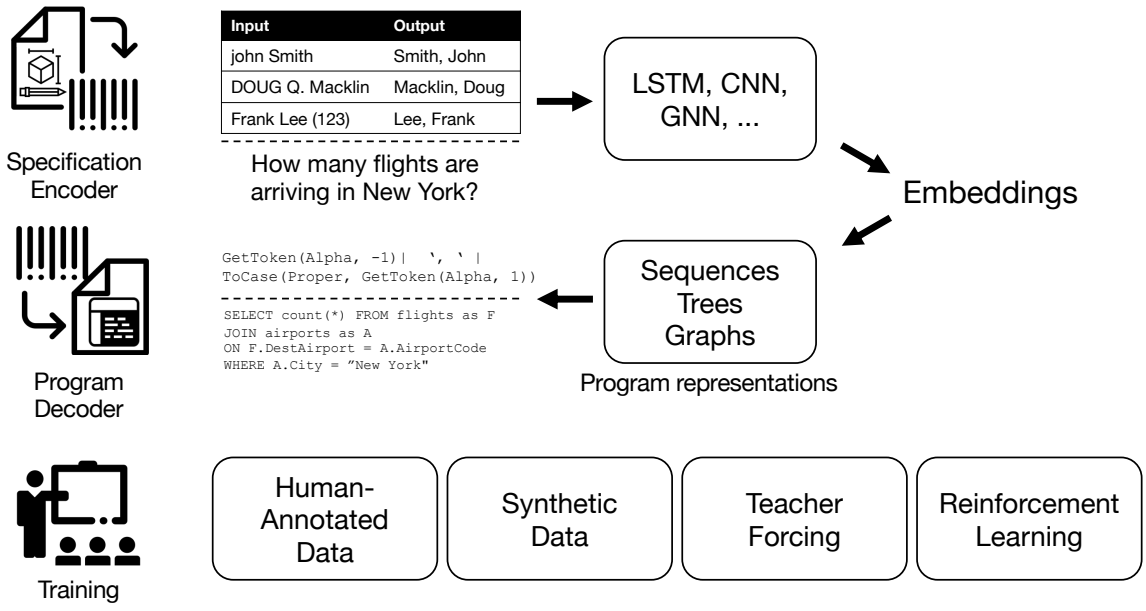


Figure 1.1: A high-level overview of the typical components involved in neural program synthesis.

vision, or graph neural networks (Li et al. 2016). This procedure transforms the specifications into a set of real-valued vectors, which we refer to as *embeddings*.

The *program decoder* then uses the embeddings to predict the code for the program. The decoder may produce the code as a sequence of words (tokens) in the target programming language, the same modality that a human programmer would use. If the program specification was also a sequence (for example, a natural language question), the overall neural architecture is a sequence-to-sequence model (Sutskever et al. 2014), which has been widely adopted for machine translation. While using token sequences as the output format is convenient, most such sequences are invalid as programs since they don't conform to the programming language's syntax. By directly generating abstract syntax trees with the decoder and forcing it to only output syntactically correct programs, we can get better results (Rabinovich et al. 2017; Yin and Neubig 2017). Further annotating the trees with extra edges about dataflow and other program properties has also seen success (Brockschmidt et al. 2018).

In order to train the neural network, past approaches have used varying data sources to learn from. With *human-annotated data*, we have many high-level program specifications and the code that human programmers wrote to fulfill those specifications. Depending on the domain and type of specification, we can instead generate large amounts of *synthetic data*: for example, we can randomly generate a program by sampling from a context-free grammar, randomly generate some input for the program, and execute the program on the input to get an output. In the end, we have an

input-output pair, and a program which we know transforms the input into the output.

1.4.2 Instantiations of Neural Program Synthesis

Parisotto et al. (2016) is an early example instantiation of neural program synthesis following the general pattern that we have identified above. They synthetically generated programs with input-output pairs, and used a program decoder that can directly generate abstract syntax trees, on the Flash Fill (Gulwani 2011) text-editing domain. RobustFill (Devlin et al. 2017b) tackled the same text-editing domain, but achieved stronger results with an LSTM (Hochreiter and Schmidhuber 1997) encoder and decoder making heavy use of attention (Bahdanau et al. 2015). When compared against the deployed Flash Fill implementation in Microsoft Excel which uses an enumerative search approach, RobustFill had a much higher accuracy when the input-output pairs contained noise (typos for example) and only slightly worse on specifications without noise.

Other papers from the same period addressed different problem domains. DeepCoder (Balog et al. 2016) generates array-manipulation programs, similar to those found in some programming contests. It uses a much simpler decoder than the others; the decoder predicts, for each of the functions in the domain-specific language, the probability it would appear in the correct program's source code. These probabilities can guide traditional search procedures and the evaluation showed that they can help find the correct program much faster. Bunel et al. (2018) developed a neural program synthesis system for Karel (Pattis 1981), an educational programming language. Karel programs manipulate an agent within a grid world, and can contain control-flow constructs like loops and conditionals, which are especially challenging for traditional program synthesis approaches. Chapters 2 and 3 of this dissertation also use Karel as an application domain.

In the natural language processing community, a long-standing problem is *semantic parsing*, or mapping natural language sentences into *logical forms* that capture the meaning of the sentences. If the sentences as high-level specifications, and if the logical forms are executable programs, we can also view this as program synthesis (with a different kind of specification than the examples described just prior). Earlier approaches to semantic parsing, such as with combinatory categorical grammars (Zettlemoyer and Collins 2005), use logical forms that closely conform to the syntactic structure of the sentence. Later works adopted a neural encoder-decoder approach that enables greater flexibility, including incorporation of other structured data as part of the input specification, and using general-purpose languages as the output logical form. For example, Ling et al. (2016) curated a dataset, from the trading card games Hearthstone and Magic the Gathering, containing natural language descriptions and other structured information from the cards, and Python/Java code that implements the cards within a game engine. Rabinovich et al. (2017) and Yin and Neubig (2017) demonstrated a neural program decoder that can directly generate abstract syntax trees, showing greatly improved results on this dataset. Similarly, the creation of large-scale datasets such as WikiSQL (Zhong et al. 2017) and Spider (Yu et al. 2018c) have spurred significant work in question answering by directly translating a natural language question into SQL, the dominant programming language used for interaction with relational databases today. In Chapter 4 of this dissertation, we will discuss an approach for generating SQL queries from natural language.

1.4.3 Neural Program Induction

The current wave of deep learning has also witnessed considerable enthusiasm for what we will refer to as *neural program induction*, following the terminology of Parisotto et al. (2016). Neural program induction uses neural networks taking inspiration for their internal structure from programs and computer architecture, incorporating components such as memory access, registers, or stacks, and are trained to solve computational problems such as sorting, addition, or graph manipulation by outputting a sequence of elementary operations. One example is the Neural Turing Machine (Graves et al. 2014), later succeeded by the Differentiable Neural Computer (Graves et al. 2016); they augment a neural network controller with a memory that it can read and write through location- and content-based addressing. Later examples include Weston et al. (2015), Sukhbaatar et al. (2015), Kaiser and Sutskever (2015), Joulin and Mikolov (2015), Kurach et al. (2016), Reed and de Freitas (2016), Cai et al. (2016), and Neelakantan et al. (2017).

These techniques use gradient descent or other optimization techniques to produce the parameters for a neural network as the final result. In contrast, the output of a program synthesis system is the code for a program in a programming language. They also require a large number of examples for any given task to train the neural network (although meta-learning techniques like used in Devlin et al. (2017a) can reduce this number). In contrast, program synthesis techniques that aim to produce a program satisfying a given specification even when the specification is very small, for example 1–5 input-output pairs. Given the similar goals, these two branches of research may converge closer together in the future, but much work remains to bridge the gap considering the fundamental differences between discrete logical forms used in programs and the continuous weights used in neural networks.

1.5 Program Synthesis and AI

1.5.1 Relationship to Other Domains

Program synthesis was a topic of study for some of the earliest researchers in artificial intelligence. For example, papers like Waldinger and Lee (1969) and Green (1969) were both presented at the first International Joint Conference on Artificial Intelligence. The emergence of deep learning has led to another convergence between the two fields today in the form of neural program synthesis. Deep learning methods, first popularized in fields like computer vision and speech recognition, are growing common as approaches for solving problems in program synthesis. Increasing capabilities in program synthesis also enable transfer in the opposite direction, as we can pose other problems under the umbrella of artificial intelligence as program synthesis problems, providing new perspectives on and approaches to them.

In natural language processing, question answering over databases can be solved by translating the questions into programs (like SQL queries) with the same intent (Finegan-Dollak et al. 2018). This approach is unsurprising since queries are the standard mode of interaction with databases. In contrast, visual question answering (Antol et al. 2015) seeks to provide natural language answers to questions about the content of an image. Considering the modality of images, it is less intuitive

that this problem would benefit from using a program as an intermediary. Nevertheless, past works like neural module networks (Andreas et al. 2016; Hu et al. 2017) successfully exploit the shared compositional aspects of programs and natural language, and tackle this problem by first predicting a program (consisting of learned neural "modules") and executing it over the image.

A central problem in computer vision is understanding the components that make up an image. With inverse graphics, we try to understand the image by inferring the instructions necessary to synthesize it. We can view this as program synthesis where the specification is the image and the instructions make up the program. In one example, SPIRAL (Ganin et al. 2018) learns to generate images like digits and faces with a sequence of brush strokes. Lopes et al. (2019) infers how to draw characters from fonts in Scalable Vector Graphics (SVG). If we broaden the definition of images to also consider three-dimensional models, Du et al. (2018), Tian et al. (2018), and Ellis et al. (2019) all address the problem of creating a CAD program that would generate a 3D object. These methods enable us to obtain a higher-level summary of images and objects from the raw data, in terms of semantically meaningful building blocks.

1.5.2 Shared Challenges

With neural program synthesis, we use tools from machine learning to perform program synthesis. While powerful, there remains many challenges with the state of the art shared between machine learning approaches in different domains. We will describe a few examples and explain how they relate to neural program synthesis.

Generalization. When we learn a model from a training dataset, or with reinforcement learning in a certain environment, we would like the model to learn the true underlying factors that explain the data. However, current methods can instead latch on to spurious correlations or quirks of a particular dataset instead. When we use the model on a test dataset, we may find that it performs much more poorly than initially expected. Perhaps the most basic failure to generalize is *overfitting* in the i.i.d. setting, where the training and test data are assumed to come from the same distribution, yet the model performs much better on the training data than the test data. In real-world use cases, this failure can be exacerbated as the training and test data distributions will differ; for example, Buolamwini and Gebru (2018) discuss the effects of skin color on gender classification using human faces.

More ambitiously, we would like our model to perform well on a new class of data points not present in the original training data even when provided with only a few or zero examples (as in few-shot learning, or zero-shot learning). For example, we would like an image classifier to be able to recognize a new animal like a zebra without hundreds or thousands of examples, as a human child might do given a few photographs or just a textual description ("four legs and has black and white stripes"). In addition, we would like models that can exploit compositional structure in the data. When if have examples images of birds with "black wings and a white belly", and "red wings and a yellow belly", we want to also understand "red wings and a white belly".

In neural program synthesis, we learn to synthesize programs from certain kinds of specifications available in the training data. We want the resulting synthesizer to work well on all kinds of specifications, even if they differ from those seen during training.

Abstraction. In order to solve large problems, humans can break them down into subproblems recursively until they become a manageable size, or solve a simplified, high-level version of the problem and then fill in the details later. When writing a long document like a book, we can first decide on the general ideas or themes in each section, then fill in the details of the subsections, paragraphs, sentences, then finally the individual words. In contrast, state-of-the-art methods in machine learning generally lack such structures. For example, while very large language models like GPT-3 (Brown et al. 2020) can generate much more coherent text than ever before, they cannot maintain consistency and structure beyond a few hundred words at best.

Considering this, much work has attempted to imbue machine learning methods with hierarchy and modularity. The use of a hierarchy has the potential to increase the interpretability of models, as they would better match our approach to problem-solving. It allows us to explicitly separate out certain parts from others, reducing the amount of data needed for learning as we can separately account for each part. If these parts appear in multiple high-level tasks, it may also help with sharing knowledge between them (for example, a cooking robot might exploit that knife chopping is necessary for many different dishes). In reinforcement learning, the options framework (Sutton et al. 1999) has been influential as a framework for adding some amount of hierarchy. In computer vision, capsule networks (Sabour et al. 2017) propose to recognize objects by first finding their parts and composing them together.

For program synthesis, we can anticipate that generating complicated programs can benefit from the use of abstraction. Programming languages contain structures like functions and classes to help developers structure their code. We also have other kinds of abstractions that do not correspond to explicit elements of a programming language, such as code idioms and design patterns. It remains a challenge for neural program synthesis methods to effectively create and use these kinds of abstractions.

Embodiment. Within a paradigm like classification, machine learning is learning a mathematical function: a mapping between two sets. However, more generally, we want to use machine learning to create intelligent agents that can have some impact upon the world. These agents cannot just think and reason disconnected from everything else; rather, they need to make observations about their environment and take actions that modify it. Robotics is an obvious example of a field where this is important, but we can find parallels in natural language processing (for agents that engage in dialogue, with humans or other agents) recommender systems, and others.

As for programs, it is important to consider that many programming languages and algorithms come in an *imperative* form, consisting of a sequence of statements that change the program's state. Furthermore, modern computers work similarly, by executing a sequence of machine instructions that modify the machine's state, and all programming languages are eventually translated into such a format for execution. There are also concurrent programs where multiple threads of execution

may modify some global state, and need to coordinate with each other; and networked programs that communicate with each other.

1.6 Contributions of the Dissertation

So far in this section, we have laid out some of the motivation for program synthesis; discussed formal methods-based, classical approaches to program synthesis and their limitations; introduced *neural* program synthesis using deep learning methods, with some example instantiations in prior work; explored how other problems in artificial intelligence can be approached through the lens of neural program synthesis; and described some core challenges in machine learning applications and their relationship to neural program synthesis.

In Section 1.4.1, we showed a high-level framework for neural program synthesis, consisting of three components: the specification encoder, the program decoder, and training. In the subsequent chapters of this dissertation, we will address one of the challenges of generalization, abstraction, and embodiment through improving one of these components in different concrete instantiations of program synthesis from input-output examples and from natural language.

- In Chapter 2, we address *embodiment* through the *specification encoder*. Specifically, we investigate how we can better synthesize imperative programs through the use of *inferred execution traces*, when we have input-output examples as the specification. An execution trace detailing each step would greatly aid a synthesizer, especially for imperative programs, but it is much simpler for users to provide input-output examples instead. We demonstrate on the Karel domain that we can gain much of the benefit of execution traces by inferring them from the input-output examples. This work was previously published as Shin et al. (2018b).

- In Chapter 3, we address *generalization* through the *training* data. For program synthesis from input-output examples, we can obtain large amounts of supervised training data through random sampling. Many current approaches achieve impressive results after training on such data. However, we empirically show that applying test input generation techniques for languages with control flow and rich input space causes deep networks to generalize poorly to certain data distributions; to correct this, we propose a new methodology for controlling and evaluating the bias of synthetic data distributions over both programs and specifications. We demonstrate on the Karel domain and a small calculator program-induction domain that training on these distributions leads to improved cross-distribution generalization performance. This work was previously published as Shin et al. (2018a).

- In Chapter 4, we address *generalization* through the *specification encoder*. We address the problem of synthesizing SQL database queries from natural language questions. As part of this, we would like our methods to generalize to domains and database schemas outside of the training set. We show how to use relation-aware self-attention (Vaswani et al. 2017; Shaw et al. 2018) within the specification encoder and obtain significant gains on the Spider

dataset (Yu et al. 2018c). This work was previously published as a preprint (Shin 2019); a later version of this work was published as Wang et al. (2020).

- In Chapter 5, we address *abstraction* through the *program decoder*. We build a a system that allows a neural program synthesizer to explicitly interleave high-level and low-level reasoning at every generation step in the decoder. It accomplishes this by automatically mining common *code idioms* from a given corpus, incorporating them into the underlying language for neural synthesis, and training a tree-based neural synthesizer to use these idioms during code generation. We apply the method to SQL generation as done in Chapter 4, and on a Python-based dataset. This work was previously published as Shin et al. (2019).

Part I

Synthesis from Input-Output Examples

Chapter 2

Synthesis with Inferred Execution Traces

In this chapter,[1] our goal is to enable better synthesis of imperative programs. Much prior work in program synthesis has focused on declarative and functional programs, such as SQL that we will address in Chapter 4 or regular expressions used in text editing. However, the most common programming languages are imperative, and many kinds of algorithms are most easily expressed imperatively, but it is especially challenging to reason about control-flow constructs like loops and conditionals.

Having access to additional structured information such as execution traces would make the synthesis easier. In an execution trace, we have access to the operation used at each step of execution, and the state of the program's environment at that time. While execution traces can provide highly detailed guidance for a program synthesis method, they are more difficult to obtain than more basic forms of specification such as input-output examples. Therefore, we use the insight that we can split the process into two parts: infer traces from input-output examples, then infer programs from traces. Through this, we achieve state-of-the-art performance on the Karel program synthesis domain.

2.1 Introduction

End-to-end neural approaches have been particularly successful in perceptual domains like computer vision where designing intermediate representations is a big challenge. In contrast, within program synthesis, there exists a great deal of structure and auxiliary information the model could learn to exploit in addition to the typically used input/output examples. An example is program execution traces, which have also been used to great effect by prior work (Kaiser and Sutskever 2015; Wang et al. 2018).

Given that an execution trace can be a strict superset of an input-output example, intuition suggests that program synthesis from execution traces should be easier than synthesis from I/O examples. Since we can replay an execution trace with the interpreter to obtain detailed information about the program state at each step, a trace-based synthesis model can rely upon the interpreter to handle the semantics of the DSL's operations, and focus more on how to infer control flow

[1] The material in this chapter is based on Shin et al. (2018b).

constructs by reconciling different paths taken in different inputs. However, execution traces are difficult to obtain as they are much more challenging for the end user to specify, so it is hard to reap their benefits.

In this chapter, we use our intuition that if encoder-decoder neural networks can synthesize programs from input-output examples, they should also be able to infer execution traces. Thus, we can split the problem into two steps: use input/output examples to infer execution traces, and then use execution traces to infer the program. Our empirical results show that this modification leads to state-of-the-art results on the Karel (Pattis 1981) program synthesis task, improving upon Bunel et al. (2018) from 77.12% to 81.3% accuracy.

Our analysis shows greater accuracy on programs of varying lengths and complexities, demonstrating the general utility of the approach. This is despite the fact that we only use straightforward maximum likelihood training, which is easier to tune than reinforcement learning methods of prior work. Nevertheless, as our method is largely orthogonal to prior techniques like reinforcement learning, our research suggests useful future directions for further improving the accuracy of neural program synthesis.

2.2 Karel Domain for Program Synthesis

$$
\begin{array}{rcl}
\text{Prog } p & := & \texttt{def main():}\, s \\
\text{Stmt } s & := & \texttt{while}(b) : s \mid \texttt{repeat}(r) : s \mid s_1; s_2 \\
& \mid & a \mid \texttt{if}(b) : s \mid \texttt{if}(b) : s_1 \texttt{else} : s_2 \\
\text{Cond } b & := & \texttt{markersPresent()} \mid \texttt{leftIsClear()} \\
& \mid & \texttt{rightIsClear()} \mid \texttt{frontIsClear()} \mid \texttt{not}(b) \\
\text{Action } a & := & \texttt{move()} \mid \texttt{turnLeft()} \mid \texttt{turnRight()} \\
& \mid & \texttt{pickMarker()} \mid \texttt{putMarker()} \\
\text{Cste } r & := & 0 \mid 1 \mid \cdots \mid 19
\end{array}
$$

Figure 2.1: The syntax of the Karel DSL as used in the dissertation. Figure from Devlin et al. (2017a).

Karel is an educational programming language (Pattis 1981), used for example in Stanford CS introductory classes and the Hour of Code initiative. It features an agent inside a grid world, where certain cells can contain *markers* or *walls* (but not both); the agent cannot enter cells where there is a wall. The agent can take the following actions:

- moving forward (`move`),

- turning left or right (`turnLeft`, `turnRight`),
- and modifying the world state by removing or adding *markers* to the current location (`pickMarker, putMarker`)

Karel programs, which are imperative, can contain branching statements (`if`, `ifElse`), `while` loops which execute as long as a condition is true, and `repeat` loops which execute for a fixed number of repetitions. The following conditions are available: whether the cell at the agent's location contains markers (`markersPresent`), and whether there are any walls nearby (`frontIsClear`, `leftIsClear`, `rightIsClear`).

Devlin et al. (2017a) introduced the use of the Karel domain for program *induction*, where a neural network learns to represent a program; in this work, we tackle Karel program *synthesis*. The goal in this domain is to learn how to generate a program in the Karel DSL given a small set of input and output grids. Formally, we are given a set of n input-output world pairs $\{(I_1, O_1), \cdots, (I_n, O_n)\}$, with some hidden program π which satisfies the property that executing π in I_1 results in O_1, I_2 results in O_2, and so on. Our task is to recover $\hat{\pi}$ by observing the n input/output pairs, such that $\hat{\pi}$ is semantically equivalent to π; in other words, $\hat{\pi}$ should have the same effect as π on any input world, but they do not need to be textually equivalent. However, note that the problem is under-specified: with n input-output pairs, it is not possible to disambiguate among all possible π, so the model needs to pick the most promising among the possibilities.

The methods in this chapter are based on Bunel et al. (2018), which applied a neural encoder-decoder approach to Karel program synthesis, similar to the work of Devlin et al. (2017b) and Parisotto et al. (2016) which was for a string-editing domain. Bunel et al. (2018) used both supervised learning with a randomly-generated synthetic dataset to train their model, as well as a reinforcement learning-based approach to further improve the model's program synthesis accuracy. As part of their work, they have developed a deep learning architecture for Karel program synthesis which we use as the basis for our approach.

2.3 Approach

2.3.1 Motivation

Past work in program synthesis, program induction, program repair, and other applications of machine learning related to programs have explored the benefits of learning using *execution traces*.

For example, in program induction, the Neural Programmer-Interpreter (Reed and de Freitas 2016) receives execution traces as supervision, and can learn complex tasks more quickly and accurately; given recursive traces, it can exhibit perfect generalization (Cai et al. 2016). Other program induction models such as the Neural Turing Machine (Graves et al. 2014) and Neural GPU (Kaiser and Sutskever 2015) have the harder task of learning directly from input/output examples, and thus need a very large amount of training data and careful hyper-parameter tuning. In program repair, Wang et al. (2018) improve upon baseline methods by learning models that use execution traces. Ellis et al. (2018b) and Ganin et al. (2018) generate execution traces for the purposes

of inverse graphics. Even for generative modeling of images, learning to generate the picture incrementally and additively (similar to execution traces in our setting) has been able to improve performance (Gregor et al. 2015).

We hypothesize that any program synthesis model must be able to internally reason about the semantics of the DSL and in particular the atomic operations. For example, in the Flash Fill system (Gulwani 2011), this knowledge is explicitly specified by the system's creators. In contrast, neural program synthesis methods need to learn both the semantics of the DSL and how to synthesize programs in the DSL entirely from the training data.

Even beyond the past work using traces and our hypothesis above, our intuition as programmers suggests that it should be easier to synthesize a Karel program given not only the input-output examples, but also the list of steps taken by the correct program to transform the input into the output. However, it is impractical to expect an end user specifying the desired program to also provide the correct execution trace for the program, since devising the trace is almost as much work as writing the actual program itself.

However, if neural networks can learn to synthesize programs from input/output examples as shown by Bunel et al. (2018) and others, it follows that they should also be able to synthesize the execution trace from input/output examples. Indeed, we can expect this to be an easier problem given that the execution trace does not contain any control flow constructs. Furthermore, as the model iteratively generates the execution trace, we can evaluate the partial trace with the Karel interpreter and provide its internal state to the model to help guide the model's next output.

Once we have a model that can recover the correct execution trace from the input/output examples for a desired program, it becomes possible to train and use a program synthesis model that takes both input/output examples and corresponding execution traces as the program specification. By including information extracted from the Karel interpreter as we run the execution trace about the current state of the world at each point in the trace, the program synthesis model has less need to internally reason about the program's semantics as some of that work is effectively offloaded to the Karel interpreter.

In the subsequent sections, we detail how we built two models to split the Karel program synthesis problem ("I/O $\rightarrow$ CODE") into two parts: I/O $\rightarrow$ TRACE, then TRACE $\rightarrow$ CODE.

2.3.2 Predicting execution traces from input/output pairs

In this chapter, an execution trace refers to an ordered set of actions: $(action_1, ..., action_T)$. In the case of Karel, actions are $move$, $turn\{Right, Left\}$, $\{put, pick\}Marker$. For a given original training example $(\pi, \{(I_1, O_1), ..., (I_N, O_N)\})$, we can generate N training examples $(I_1, O_1, (action_1, \cdots, action_T)_1), \ldots, (I_N, O_N, (action_1, \cdots, action_T)_N)$ for trace prediction by running π on these I/O pairs and recording the actions taken by the program. Thus, if the original training data contained K examples where each example contained N I/O pairs, then we obtain a I/O pair to trace dataset of $K \cdot N$ examples suitable for supervised learning.

Figure 2.2 shows the deep learning model architecture we used for this task. To encode the input/output examples, we use a convolutional neural network with a final fully-connected layer, taken from Bunel et al. (2018). To generate the sequence of actions, we use a two-layer LSTM

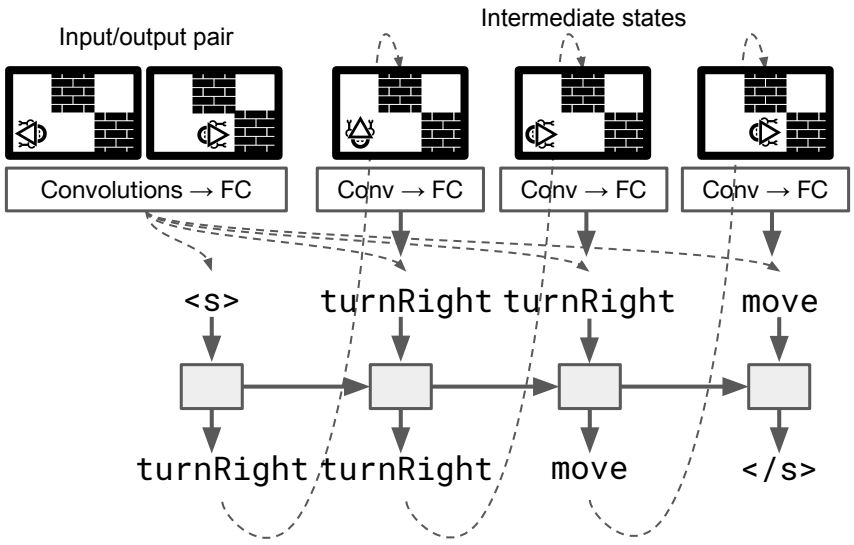


Figure 2.2: Architecture of I/O → TRACE model.

decoder. At each step of the decoder, it receives as input the concatenation of the following: 1) an embedding of the action taken in the previous step, 2) the input/output pair embedding, and 3) an embedding of the current state of the grid after executing all of the past actions. In theory, the LSTM can learn to keep track of the current state of the grid internally, rendering the third input unnecessary; however, as we will see in Section 2.4.4, explicitly using the Karel interpreter to track the current grid state helps the model better understand how the grid is changing after each action and maintain more context.

2.3.3 Synthesizing programs from input/output examples and execution traces

To create a model which uses both a set of input/output examples and execution traces for generating the desired program, we started with the architecture from Bunel et al. (2018) and extend it to also take the execution trace as an input. Specifically, we add a bidirectional LSTM to the architecture which is responsible for producing an embedding of the execution trace for each step in the trace.

Given an execution trace $(action_{1,n}, ..., action_{T,n})$ and an initial state $state_{1,n} := I_n$ for the nth input/output example, we can use the Karel interpreter to replay the actions to obtain $state_{2,n}, \cdots, state_{T+1,n}$. For Karel, each state contains the full grid world and the objects within it: the location of the agent, its orientation, the current number of markers in each cell, and the locations of walls (which cannot be manipulated by the agent and therefore do not change through the course of execution). Given the mismatch in lengths, and also given that the actions semantically occur in between the states, we interleave the two to create an input of length $T + (T + 1) = 2T + 1$:

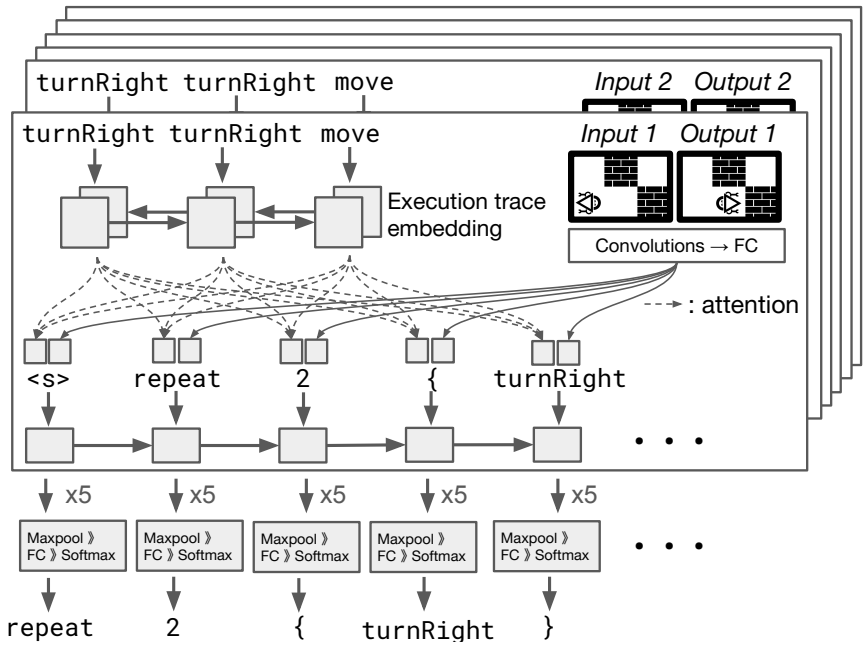


Figure 2.3: Architecture of TRACE → CODE model. We follow the architecture in Bunel et al. (2018), but add an execution trace input.

$(state_{1,n}, action_{1,n}, state_{2,n}, \cdots, action_{T,n}, state_{T+1,n})$.

To provide this input to the bidirectional LSTM, we embed $state_{t,n}$ and $action_{t,n}$ in the following way. For each state, we evaluate the four conditionals (`markersPresent`, `frontIsClear`, `leftIsClear`, `rightIsClear`) that can influence the program's flow of execution, embed each boolean value and concatenate them. For each action, we look up the corresponding embedding from a table. Both sets of embeddings are randomly initialized and learned. We will denote the $2T + 1$ outputs for the nth trace from the bidirectional trace LSTM as $\mathcal{T}_{1,n}, \cdots, \mathcal{T}_{2T+1,n}$.

We also tried a variant where the input consists of $T + 1$ elements. First, we append appending a final `</s>` to the list of actions: $action_{T+1,n} =$ `</s>`, to make the lengths match. We then embed each $state_{t,n}$ and $action_{t,n}$, and then concatenate the embeddings together. In addition to the conditional values, we also tried providing an embedding of the grid itself to the LSTM, using a similar convolutional neural net as used to encode each grid in the I/O → TRACE model. However, we found that both variants were inferior compared to the method described in the previous paragraph.

Following Bunel et al. (2018), we also encode each input/output example using a convolutional neural network with a final fully-connected layer. We generate the program one token at a time with a decoder LSTM. As in Bunel et al. (2018) and Devlin et al. (2017b), we run a separate LSTM for each input/output pair; the LSTMs have separate states but shared weights. At decoding step i, the LSTM for the nth input/output example receives the concatenation of following:

- an embedding of the token generated in step $i-1$, or of <s> at step 0 (the start of decoding)
- the input-output pair embedding for (I_k, O_k),
- the *context vector* $c_{i-1,n}$ for step $i-1$: a weighted sum of $\mathcal{T}_{1,n}, \cdots, \mathcal{T}_{2T+1,n}$, computed using a multiplicative attention mechanism based on $o_{i-1,n}$, the LSTM's output at step $i-1$. Note that $c_{0,n} = \mathbf{0}$.

We obtain the decoder LSTM output $o_{i,n}$ for each of the N input/output pairs, compute the context vector $c_{i,n}$ as described above, and concatenate them to obtain $\tilde{o}_{i,n} = \text{Concat}(o_{i,n}, c_{i,n})$. To compute the logits over the ith program token, we compute $W \cdot \text{MaxPool}(\tilde{o}_{i,1}, \cdots, \tilde{o}_{i,N})$, where $W \in \mathbb{R}^{v \times d}$, and v is the size of the output vocabulary. See Figure 2.3 for a visual depiction of the overall architecture.

To train this model, we tried two different sources of supervision. Recall that each entry in the provided training data consists of a program π and 5 input-output pairs $\{(I_1, O_1), ..., (I_5, O_5)\}$). First, we can execute π on $I_1, \ldots, I_5$ to obtain the execution trace on each input; we refer to this trace as the *gold* trace. However, we will not have access to the gold trace when we wish to actually use this model for program synthesis from input/output examples.

Second, we can use the I/O $\rightarrow$ TRACE model from Section 2.3.2 to infer a valid trace for the given input-output pair; we refer to this trace as the *inferred* trace. Unfortunately, this model does not always succeed at recovering a correct trace for the input-output pair, in which case we substitute a trace containing a single `UNK` action and two grid states, the input grid and the output grid. Furthermore, the trace may deviate from the actions taken by the true program even if the final state is identical, as certain actions can be permuted without any effect on the output (such as `turnLeft` and `pickMarker`), and certain sequences of actions (such as `turnLeft` then `turnRight`) are no-ops. Nevertheless, we will only have access to the inferred trace at inference time, so it is useful to match the training and test distributions more closely.

2.4 Experiments

2.4.1 Training dataset and procedure

To train and test our models, we used the same dataset as Bunel et al. (2018), from https://bit.ly/karel-dataset. The training dataset consists of 1,116,854 entries, and the test dataset contains 2,500 entries. Each entry in the dataset contains a Karel program and 6 input-output pairs which satisfy that program. For training the I/O $\rightarrow$ TRACE model, we used all 6 input-output pairs within each entry for a total of 6,701,124 training traces. For training the TRACE $\rightarrow$ CODE model (and our reimplementation of the I/O $\rightarrow$ CODE model from Bunel et al. (2018)), we randomly sample 5 out of the 6 input/output examples (and corresponding traces) each time we sample an entry from the training data. In general, we endeavored to follow the training regime from the prior work as closely as possible, although we discovered that SGD with gradient clipping worked better than Adam for training the models. For all of the evaluations of TRACE $\rightarrow$ CODE we used beam search with size 50.

Given that each program in the test data comes with 6 input-output pairs, but we use 5 of them for specifying the program to the model, so only 1 input-output pair remains for purposes of testing whether the predicted program is semantically equivalent to the ground truth program. To enable more thorough testing, we used the method described in Chapter 3[2] to generate 99 more input/output pairs for each of the 2,500 test programs.

2.4.2 Performance metrics

When evaluating the model, we use beam search both to get outputs that have higher log likelihood than what can be obtained with greedy decoding, and also to obtain multiple candidate sequences. As such, we use multiple criteria to report the performance of the models.

First, for purposes of comparison, we have the same metrics as used by the prior work: **Top-K Exact Match**, which measures how often one of the top K output programs of the model textually matches the original program exactly; and **Top-K Generalization**, which denotes the fraction of test instances for which one of the top K output programs will have the correct behavior across the 5 input/output examples used to specify the program to the model, as well as the held-out 6$^{\text{th}}$ input/output example.

As an alternative to these metrics, we suggest to use what we call **Top-K Model-Guided Search Accuracy**. In this metric, we consider the top K program outputs in order, from most likely to least likely. We test each candidate program on the 5 input/output examples that specify the program, and see if it works correctly on those 5. We return the first such program (the top-ranked one) as the solution, and then test it on the held-out 6$^{\text{th}}$ program to report the accuracy. The motivation for this approach is two-fold. First, we already have the 5 I/O examples to specify the program for the model to produce, and so we might as well use them to filter any unsatisfactory outputs of the model, to reap the benefits of having a precisely checkable specification for the correct answer. Second, this metric is more comparable to other methods in the literature that use a search-based method for program synthesis, either with handwritten heuristics or with machine learning models (such as (Balog et al. 2016)); indeed, such methods will often try thousands or millions of candidate programs, rather than the comparatively small $K = 50$ which we used for our experiments.

We also define additional metrics that use a larger number of input/output pairs to test a candidate program. **Top-K Generalization (N test input/output pairs)** measures the fraction of test instances for which one of the top K output programs performs correctly on the 5 specifying input/output examples as well as on all of N additional held-out input/output examples. **Top-K Model-Guided Search Accuracy (N test input/output pairs)** is a similar modification to Top-K Model-Guided Search Accuracy, which uses N held-out examples rather than just one. Note that, if we set $N = 1$, then these metrics are the same as the ones defined in the earlier paragraphs. Increasing the number of test cases also diverges from the evaluation methodology of previous work, but reduces the amount of false positives on the generalization measure. However, when N is large, this metric may also be overly pessimistic; there may be many reasonable Karel programs which

[2]Specifically, we followed the method in Section 3.3.2 but we attempted to simulate the observed empirical distribution for the number of cells with markers, the number of cells containing walls, and the number of markers in each marker-containing cell.

Table 2.1: Comparison of our best model with previous work from Bunel et al. (2018). "Gen." stands for generalization accuracy.

	Top-1		Top-50	
	Exact Match	**Gen.**	**Guided Search**	**Gen.**
MLE (Bunel et al. 2018)	39.94%	71.91%	–	86.37%
RL_beam_div_opt (Bunel et al. 2018)	32.17%	77.12%	–	85.38%
I/O → CODE, MLE	40.1%	73.5%	84.6%	85.8%
I/O → TRACE → CODE, MLE	**42.8%**	**81.3%**	**88.8%**	**90.8%**

Table 2.2: Evaluation of I/O → CODE and I/O → TRACE → CODE with varying numbers of test cases.

Test I/O pairs	Method	**Top-1 Gen.**	**Top-50 G.S.**	**Top-50 Gen.**
N = 1	I/O → CODE	73.5%	84.6%	86.2%
	I/O → TRACE → CODE	81.8%	88.5%	90.7%
N = 50	I/O → CODE	56.8%	63.0%	72.8%
	I/O → TRACE → CODE	62.6%	66.0%	77.0%
N = 100	I/O → CODE	55.6%	61.7%	71.7%
	I/O → TRACE → CODE	61.3%	64.6%	75.6%

satisfy the 5 input-output pairs used to specify the desired program, but the model may have picked one different from the ground truth program which also has different behavior on some inputs.

2.4.3 Evaluation of I/O → TRACE → CODE

In Table 2.1, we compare our best I/O → TRACE → CODE model (created by gluing together I/O → TRACE and TRACE → CODE) against the previous work of Bunel et al. (2018). We reimplemented their MLE model (labeled as I/O → CODE), obtaining slightly better results compared to theirs.

We note that we did not implement the RL_beam_div_opt training method of Bunel et al. (2018), and so our results are all based on MLE training. Nevertheless, our I/O → TRACE → CODE method outperforms all others on all metrics, including the best result in Bunel et al. (2018). We anticipate that using reinforcement learning methods (such as RL_beam_div_opt) can improve our method's accuracy even further.

In Table 2.2, we compare our reimplementation of Bunel et al. (2018) with our method. As in the previous table, our method I/O → TRACE → CODE is ahead of I/O → CODE on all metrics. When we increase N to 100, our method's top-1 generalization and top-50 guided search accuracy

Table 2.3: Comparing performance on different slices of data.

Slice	% of dataset	I/O → CODE	I/O → TRACE → CODE	Δ%
No control flow	26.4%	100.0%	100.0%	+0.0%
Only Conditions	15.6%	87.4%	91.0%	+3.6%
Only Loops	29.9%	91.3%	94.3%	+3.0%
With all control flow	73.6%	79.0%	84.8%	+5.8%
Program length 0-15	44.8%	99.5%	99.5%	+0.0%
Program length 15-30	40.7%	80.8%	86.9%	+6.1%
Program length 30+	14.5%	48.6%	61.0%	+12.4%

Table 2.4: Evaluation of I/O → TRACE models.

	Top-1		Top-10	
	Exact Match	Correct	Exact Match	Correct
No grids	**58.7**%	92.5%	**59.3**%	95.9%
LGRL	57.6%	94.8%	58.0%	97.4%
PRESNET	57.8%	**95.2%**	58.2%	**98.0%**

declines by about 25% relative (17.9 and 22.9 percentage points respectively), whereas top-50 generalization accuracy declines by about 17% relative. We can see that when N is large, guided search does not help as much, since the model will often predict a program which is correct on the 5 specifying input/output pairs but which fails to generalize.

We also analyzed how models performed on various slices of the test data in Table 2.3: programs with no control flow (only actions); programs with conditionals (`if` or `ifElse`) but not loops (`repeat` or `while`); programs with loops but no conditionals; and programs containing at least one control flow element. We also partitioned the data depending on the length of the gold program into three buckets.

We can observe that I/O → TRACE → CODE improves upon I/O → CODE within every slice of the data. The magnitude of the improvement is most significant on long programs, which provides supporting evidence for our hypothesis in that the I/O → CODE model would need to internally keep track of the Karel state but have trouble doing so.

2.4.4 Evaluating I/O → TRACE and TRACE → CODE separately

For the first part of our approach (I/O → TRACE), in Table 2.4 we show results of predicting the 5 execution traces from the 5 input/output examples used to specify a Karel program synthesis task. In this table, we consider a result to be correct if all 5 predicted traces transform the corresponding

Table 2.5: Evaluation of TRACE → CODE models.

Train traces	Test traces	**Exact Match**	**Correct**	**Guided Search**
Gold	Inferred	39.2%	76.5%	81.8%
Inferred	Inferred	**42.8%**	**81.3%**	**88.8%**
Gold	Gold	54.0%	86.4%	92.4%

input states to the output states when executed in the Karel interpreter. As discussed in Section 2.3.2, there exists many possible execution traces which transform a given input state to the output state; therefore, the exact match accuracy is much lower than the correctness metric.

The LGRL model uses the same architecture for convolutional encoder as in Bunel et al. (2018). We also augmented it with residual connections between layers, results for which are reported as PRESNET model. Furthermore, to confirm that the interpreter's current state helps the I/O → TRACE model produce correct traces, we have trained a variant ("No grids") which omits the inputs of the grid state.

For the second part (TRACE → CODE), Table 2.5 compares a model trained on *gold* traces against one trained on *inferred* traces from the best I/O → TRACE model. Due to the distributional differences between the gold and inferred traces, the model trained on gold traces does poorly on inferred traces.

We also tried an evaluation using the gold execution traces from the test set. As discussed earlier in Section 2.3.3, the gold execution traces would not normally be available at test time, so this evaluation serves as a hypothetical comparison against our main result.

2.5 Related Work

Program synthesis from examples. There have been several practical applications of programming by example based on search techniques and carefully crafted heuristics, such as Gulwani (2011). More recent work has started to apply deep learning for program synthesis from examples such as RobustFill (Devlin et al. 2017b), DeepCoder (Balog et al. 2016), Neuro-Symbolic Program Synthesis (Parisotto et al. 2016), and Deep API Programmer (Bhupatiraju et al. 2017). Gaunt et al. (2016) provides a comparison of various program synthesis from examples approaches on different benchmarks, showing limitations of existing gradient descent models. Bunel et al. (2018), which also uses the domain of synthesizing Karel programs from examples, learns to predict the correct program with a deep learning model by leveraging the syntax constraints of the program language and training via reinforcement learning to generate more consistent programs.

Leveraging interpreters for inverse graphics. In recent years, there has been work on learning semantics of interpreters for inverse graphics with neural networks: learning how to control a

drawing engine to reproduce a given picture, or in other words, recovering its underlying structure. In Ellis et al. (2018b), the authors first infer the execution trace of a drawing program and leverages a generic program search algorithm on the trace. Ganin et al. (2018) instead simultaneously uses techniques from reinforcement learning and adversarial training to teach an agent how to generate a program which renders the desired image. These methods provide evidence that having explicit prediction of traces or steps aids in learning the semantics of an interpreter, which is an important component of program synthesis.

Execution-guided program synthesis. Concurrent with and after the publication of Shin et al. (2018b), other work has used program states and interpreters in program synthesis. Zohar and Wolf (2019) synthesize array-manipulation programs (similar to Balog et al. (2016)) by using a neural network to repeatedly predicting the next statement in the program given the current state, and executing it to obtain a new state. Ellis et al. (2019) takes a similar approach but within an MDP formalism, using string-editing programs and CAD programs for building 3D models as evaluation domains. However, neither of these methods generate programs with loops or conditionals. Chen et al. (2018) also generates programs for Karel, including loops and conditionals, by sequentially predicting the next statement given the current state.

2.6 Discussion

From our results, we consider confirmed our hypothesis that it is beneficial to use traces for explicitly training a model that learns the semantics of interpreter, separately from the task of synthesizing the code for the correct program.

Interestingly, the predicted trace and gold trace fail to match exactly in half of the cases even though the predicted trace is correct. Indeed, the TRACE $\rightarrow$ CODE model trained on gold traces doesn't perform as well on inferred traces. That said, when we evaluated TRACE $\rightarrow$ CODE on gold traces at validation time, it outperformed the model trained on inferred traces. Since I/O $\rightarrow$ TRACE independently predicts traces for each input-output pair, we hypothesize that they lack consistency with each other compared to the gold traces. Thus, for future work we suggest to investigate inferring the execution trace for a given input-output pair, conditioned on already generated execution traces for the same underlying program but on different input-output pairs. We also noticed that the TRACE $\rightarrow$ CODE model trained on predicted traces performed much worse when evaluated on gold traces compared to predicted traces. This phenomenon suggests that training a TRACE $\rightarrow$ CODE model with multiple options of traces sampled from I/O $\rightarrow$ TRACE and from gold traces may improve the model's resilience and further improve the accuracy.

We also leave for future work exploring usage of reinforcement learning objectives (similar to Bunel et al. (2018)) for the training of these models. We see two possible ways of applying these objectives: training I/O $\rightarrow$ TRACE and TRACE $\rightarrow$ CODE models separately with the reward of passing unseen tests; and training I/O $\rightarrow$ TRACE and TRACE $\rightarrow$ CODE models jointly end-to-end, where traces are decoded into symbolic form and reinforcement learning allows propagation of learning signals between the two parts.

Chapter 3

Synthetic Datasets for Neural Program Synthesis

In the previous chapter, we demonstrated a method to synthesize imperative programs by first predicting the execution trace that the program should take from input-output examples, and using it to predict the program itself, including loops and conditionals. To train the two I/O $\rightarrow$ TRACE and TRACE $\rightarrow$ CODE models which perform these tasks, we need a large amount of training data for supervised learning. We used the same dataset from prior work containing more than a million training instances, where each instance contains input-output pairs and a correct program for those pairs.

In many applications of machine learning, having access to a large amount of high-quality data is critical for success. For neural program synthesis from input-output examples, we are fortunate in that we can generate large amounts of training data synthetically; first randomly generate the code for a program, then some inputs suitable for this program, and execute the program on these inputs to obtain corresponding outputs. Since we can repeat this to generate as much data as much as we like, we might hope that we will avoid typical problems caused by insufficient data, such as overfitting and failure to generalize to new examples. In this chapter, we empirically demonstrate that the precise methodology used is critical.

First, we show that a model trained on an existing dataset, with about 40% accuracy on the corresponding test set, fails when we re-sample the input-output pairs from various "narrow" distributions. Then, when we re-generate and re-train the model on training data where we explicitly ensure uniformity on various *salient random variables*, the accuracy recovers almost completely. Furthermore, when we test on a small challenge test set consisting of difficult real-world problems, we can improve accuracy by about 1.75$\times$ (11.1% to 19.4%) by training on this new training set.

[1] The material in this chapter is based on Shin et al. (2018a).

3.1 Introduction

In neural program synthesis and neural program induction, as large manually-curated datasets do not exist, the typical approach is to train models on large synthetically generated datasets. Presumably, if a model can accurately predict *arbitrary* program outputs (for induction) or programs in the DSL (for synthesis) then it has likely learnt the correct algorithm or DSL semantics.

Although this approach has led to some impressive synthesis results in many domains, synthetically generating datasets that cover all DSL programs and the corresponding input space can be problematic, especially for more complex DSLs like *Karel the Robot* (Pattis 1981) which includes complex control-flow primitives (while loops and if conditionals) and operators. Likewise, for induction tasks, the sampling procedure for program specifications may lead to undesirable biases in the training distribution that inhibit strong generalization.

In this chapter, we consider two problem settings. The first is the Karel domain and the program synthesis model from Bunel et al. (2018). We identify many distributions of input examples and DSL programs for which the Karel synthesis model performs poorly. The second problem setting is a program induction problem in which a model is trained to execute and predict the output of simple arithmetic expressions, which we denote the *Calculator* domain; for this domain, we considered common synthetic data generation strategies including one from `tensor2tensor` (Vaswani et al. 2018), an open-source deep learning library. Upon analysis, we find evidence of undesirable artifacts resulting from certain biases in the generation algorithm.

Our results indicate that models trained with common methodologies for synthesizing datasets fail to learn the full semantics of the DSL, even when they perform well on a test set, and suggest the need of a more principled way to generate synthetic datasets. For some program and input distributions, the state-of-the-art neural synthesis models perform quite poorly, often achieving less than 5% generalization accuracy. We develop a new methodology for creating training distributions over programs in the DSL to mitigate some of these issues. Moreover, unlike previous works that have ignored considering the distributions over input space, we show that input distributions also play a significant role in determining the synthesizer performance. Our methodology involves defining the distribution over DSL programs and input space using a set of random variables to encode much of the valuable features which describe the data, e.g. in the Karel domain, the amount of control flow nesting in programs or the number of markers present in the inputs.

Our methodology allows us to identify several specialized distributions over the input space and Karel programs on which the neural program synthesis models (Bunel et al. 2018) perform poorly when trained on traditional program and input distributions, and tested on our new distributions. From this, we design new training distributions by ensuring greater uniformity over the random variables in our methodology. By retraining the same architecture on these new training data distributions, we observe a greater ability to generalize, with significant improvements when evaluated on the aforementioned test sets. We also observe similar improvements in the *Calculator* domain as well.

This chapter makes the following key contributions:

- We propose a new methodology to generate different desirable distributions over the space of

datasets for program induction and synthesis tasks.

- We instantiate the methodology for the Karel and Calculator domains and show that model generalization is worse on datasets generated by our technique.
- We then retrain models in both domains and demonstrate that models achieve greater overall generalization performance when trained on datasets generated with our methodology.

3.2 Data Generation Methodology

Currently, automated data generation focuses in large part on a constructive process, whose parameters can be tuned. We propose a complementary approach in which we perform a subsequent filtering step on this process to ensure that the resulting distribution has certain properties.

We define a *salient* random variable as one whose distribution in the final dataset is of interest. In the case of program synthesis, we consider two kinds of salient variables: variables denoting important features of a program in the given DSL, such as its length and degree of nesting; and variables denoting features for the input space.

In many cases, we can modify our sampling procedure to ensure a desirable distribution of a particular salient variable. However, for some salient variables, it is infeasible to tune the parameters of a given sampling procedure in order to obtain a desired distribution for that salient variable. For example, if we sample programs directly from a context-free grammar, it is difficult to control the distribution of various salient variables such as program length, degree of nesting, etc. This is a notable problem in both the Karel and Calculator domains.

Furthermore, within the context of program synthesis specifically, there is often an additional challenge: not all inputs are valid for all programs. For example, in the Karel domain, an input for a given program would be invalid if the program attempts to perform illegal actions for that input (such as `move` into walls or `pickMarker` in a cell containing no markers). The requirement that the program/input pairs suit each other itself acts as an unpredictable filter that makes it difficult to ensure uniformity of salient variables by tuning generation parameters.

As such, we propose a methodology for randomly sampling a dataset $\mathcal{D}$ (consisting of elements of $\mathbb{S}$) while ensuring that a given salient variable $\nu : \mathbb{S} \to \mathbb{X}$ (where $\mathbb{X}$ is finite and discrete), denoted as a random variable $X = \nu(s)$, has a uniform distribution throughout $\mathcal{D}$. To do this, we first sample an example $s \sim q(\cdot)$ from an original distribution q. We then add s to $\mathcal{D}$ with probability $g(s)$, where

$$g(s) = (P_q[X = \nu(s)] + \varepsilon)^{-1} \left(\min_{x \in \mathbb{X}} P_q[X = x] + \varepsilon \right),$$

with $P_q[X]$, the probabilities induced over X via q, calculated empirically based on counts computed with past samples drawn from q. We repeat the above until $\mathcal{D}$ is of a desired size. For full pseudocode see Section A.1.1 in the appendix.

We use $\varepsilon \in \mathbb{R}^+$ as a hyperparameter to trade off the runtime of the above procedure with the level of X's uniformity in $\mathcal{D}$. In Section A.1.2 (in the Appendix), we provide a probabilistic bound on the uniformity of X in the resulting distribution, for the case where $\varepsilon = 0$. Also, for when $\varepsilon > 0$,

$$\begin{gathered}\texttt{GridWidth}: m \\ \texttt{GridHeight}: n \\ \texttt{Markers}: \{(i,j,k)_l\} \\ \texttt{Walls}: \{(i,j)_t\} \\ \texttt{KarelLoc}: (i,j) \\ \texttt{Orientation}: d \in \{\texttt{N},\texttt{S},\texttt{E},\texttt{W}\} \\ 2 \leq m,n \leq 16; i \leq m; \\ j \leq n; 1 \leq k \leq 9\end{gathered}$$

Figure 3.1: A declarative specification of the space of valid input worlds for Karel programs.

we can show that drawing a single sample is possible in $O(\frac{1}{\varepsilon})$ calls to the original sampler (proof in Section A.1.3, with empirical experiments in Section A.1.4 and A.1.5). Increasing ε increases the algorithm's speed, at the cost of allowing the distribution of X in $\mathcal{D}$ to further diverge from uniform.

3.3 Application to Karel: Experiments with New Test Distributions

We use the same domain of Karel from Section 2.2, with the same set of statements and control-flow structures, and the same problem definition for program synthesis from $n = 5$ input-output pairs. Recall that in Karel, the program controls an agent which lives in a rectangular grid world; here, we further detail the the space of valid input worlds to Karel programs in Figure 3.1. As with Chapter 2 and Bunel et al. (2018), we assume a bound on the input grid size to be $2 \leq m, n \leq 16$. Each cell (i, j) in a grid can either be empty, contain an obstacle (i.e. a wall, specified by the list `Walls`), or contain $k \leq 9$ markers (defined using the list `Markers`). The agent starts at some cell denoted by `KarelLoc` in the grid (which may contain markers but no obstacle) with a particular orientation direction denoted by `Orientation`.

In this section, we instantiate our abstract data generation methodology in Section 3.2 specifically for Karel to generate different test datasets. By imposing a more uniform distribution over the salient random variables when generating the input-output specifications and target programs which make up the test set, we observe much lower accuracies compared to the original test set.

3.3.1 Salient Variables in Karel

We devised the following salient random variables to describe the input space in Karel:

- *Grid size*: Dimensions of the grid in which Karel can act.
- *Marker ratio*: Fraction of cells with at least one marker.
- *Wall ratio*: Fraction of cells which contain a wall.

- *Marker count*: Number of markers that are present in a cell containing markers.
- *Number of grids*: Number of input-output pairs shown to the model to specify the desired program.

For the program space in the Karel DSL, we consider the following random variables:

- *Program size*: Size of the program in terms of number of tokens.
- *Control flow ratio*: Number of control flow structures appearing in the program.
- *Nested control flow*: The amount of control flow nesting in programs (e.g. while inside if).

3.3.2 Changing the Input-Output Distribution

We reproduced the encoder-decoder model of Bunel et al. (2018) and trained it using the provided synthetic training set with the teacher-forcing maximum likelihood objective. On the existing test set, our model achieves 73.52% generalization accuracy, slightly higher than the 71.91% accuracy reported in Bunel et al. (2018). *Generalization accuracy* denotes how often the model's output is correct on both the 5 input-output examples shown to the model and the remaining held-out 6$^{\text{th}}$ input-output example.

To test how the model may be sensitive to changes in the input-output examples used to specify the program, we created new test sets by sampling new input grids and running them on each of the programs in the existing test set to obtain new input-output pairs. By keeping the programs themselves the same, we avoid inadvertent changes in the inherent difficulty of the task (the complexity of the programs to be synthesized).

Salient random variables with uniform distribution. We first generated grids such that they would follow a distribution that is as uniform as possible in the salient features in Section 3.3.1. We used the following procedure to sample each grid: 1) sample the *grid size* (height and width) from $x, y \sim \mathcal{U}\{2, \dots 16\}$; 2) sample the *marker ratio* $r_{\text{marker}} \sim \mathcal{U}(0, 1)$ and *wall ratio* $r_{\text{wall}} \sim \mathcal{U}(0, 1)$; 3) for each cell $(i, j), 0 \leq i < x, 0 \leq j < y$ in the grid, sample $m_{i,j} \sim Bernoulli(r_{\text{marker}})$ and $w_{i,j} \sim Bernoulli(r_{\text{wall}})$; 4) if $m_{i,j} = 1$ and $w_{i,j} = 0$, sample *marker count* $mc_{i,j} \sim \mathcal{U}\{1, \dots 9\}$, otherwise set $mc_{i,j} = 0$; 5) place walls and markers in grid according to $w_{i,j}$ and $mc_{i,j}$; 6) place Karel at a random location (not containing a wall) and with a random orientation. After generating 5 input grids for a given program, we ensure that the program does not crash on any of them and also check whether the 5 input grids exhibit complete *branch coverage* (i.e., each branch is taken by at least one of the 5 inputs). If either of these conditions are not satisfied, we discard all 5 grids and start over.

On this dataset, the model trained on existing data achieved generalization accuracy of only 27.9%, which was a drop of 44.6pp from the existing test set's generalization accuracy of 73.52%.

Table 3.1: Generalization accuracies of the baseline model and a model trained on a uniform input-output distribution, on selected datasets. G, U, and A stand for $Geom(0.5), \mathcal{U}\{1, \dots 9\}$ and $10 - Geom(0.5)$ respectively. See Section 3.3.2 for dataset generation details, and Section 3.4.1 for details about the Uniform model.

r_{wall}		0.05			0.25			0.65			0.85	
r_{marker}		0.85			0.65			0.25			0.05	
$\mathcal{D}_{\text{marker count}}$	G	U	A	G	U	A	G	U	A	G	U	A
Baseline (%)	24.30	1.32	0.04	21.08	2.98	0.08	16.63	13.31	6.63	15.99	12.88	12.98
Uniform (%)	69.37	70.21	68.99	63.25	63.74	62.78	65.83	67.39	68.09	77.32	78.63	80.19
Δ	+45.07	+68.89	+68.95	+42.17	+60.76	+62.70	+49.20	+54.08	+61.46	+61.33	+65.75	+67.21

Salient random variables with narrow distributions. We further investigated the performance drop noted above by synthesizing "narrower" datasets that captured different parts of the joint probability space over the salient input random variables. For each narrow dataset, we selected r_{wall} and r_{marker} (both between 0 and 1) as well as a distribution $\mathcal{D}_{\text{marker count}}$ which would be the same for all I/O grids. Then, we follow the procedure below for each grid: 1) sample the *grid size* (height and width) $x, y \sim \mathcal{U}\{10, \dots 16\}$; 2) randomly choose $xy \cdot r_{\text{wall}}$ cells to contain walls, and $xy \cdot r_{\text{marker}}$ cells for markers; 3) sample $mc_{i,j} \sim \mathcal{D}_{\text{marker count}}$ for all cells (i, j) chosen to contain markers; 4) place Karel at a random location (not containing a wall) and with a random orientation. In our experiments, we primarily used 3 different distributions for $\mathcal{D}_{\text{marker count}}$: $Geom(0.5)$ truncated at 9, $\mathcal{U}\{1, \dots 9\}$, and $10 - Geom(0.5)$ which, when sampled, has a value equal to 10 minus a sample from Geom(0.5), truncated at 1.

The results are shown in Table 3.1 (row 1, "Baseline (%)"). We discovered that the most correlated factor with model performance was the distribution $\mathcal{D}_{\text{marker count}}$. A more negative skew consistently lowered model performance, and this effect was more pronounced at higher values of r_{marker}.

3.3.3 Changing the Program Distribution

We will now examine how the existing model can surprisingly fail to perform well at synthesizing certain programs that are different from those in the existing validation and test sets.

Performance on complex DSL constructs. We examined whether or not the model could succeed in synthesizing programs which require nesting of conditional constructs. This was of interest since these programs were relatively rare in the training dataset. We generated an evaluation dataset comprised solely of programs that contained `while` inside `while` statements, and another dataset in which all programs had `while` inside `if` statements.[1] We found that the model fared very poorly on these datasets, achieving only 0.64% and 2.23% accuracy respectively.

[1] To avoid any negative effects from changes in the input-output distribution, we attempted to ensure that $r_{\text{wall}}, r_{\text{marker}}$ and $\mathcal{D}_{\text{marker count}}$ matches that of the provided training and test sets.

Table 3.2: Results on programs only containing actions. The generalization accuracy on action-only programs in the existing test set is 99.24%. See Section 3.4 for details on Action-Only Augmented.

	Program length							
Model type	1	2	3	4	5	6	7	8
Baseline	16.00%	30.00%	44.24%	52.88%	56.56%	66.94%	67.16%	73.06%
Action-Only Augmented	20.00%	41.60%	52.24%	61.72%	63.04%	72.20%	72.74%	78.12%

Programs only containing actions. Intuitively, much of the difficulty in the Karel program synthesis task should come from inferring the control flow statements, i.e. `if`, `ifElse`, and `while`. Synthesizing a Karel program that only contains actions is intrinsically a much more straightforward task, which a relatively simple search algorithm (such as A*) can perform well.

We performed an experiment using test datasets generated by enumerating action-only programs of various lengths. As there are five actions (`move`, `turnLeft`, `turnRight`, `putMarker`, `pickMarker`), there exist 5^L textually unique action-only programs for length L. We sampled up to 500 unique programs of lengths $1, 2, \ldots, 8$. For each program, we generated 10 specifications, each containing 5 input-output pairs. We sampled each input-output pair from the set of all input grids in the existing training data (of which there are 6.7 million), as to match its distribution as closely as possible.

Table 3.2 shows the results. Remarkably, even though the underlying programs have relatively low complexity, the model's accuracy is lower on every one of these action-only test sets than the existing provided test set. The generalization accuracy grows as the program length becomes longer, even though those programs should be harder to synthesize.

Among the existing action-only programs in the test set, the model's generalization accuracy on that subset is 99.24%. Given the surprising nature of this result, we investigated the difference between the action-only programs we generated, and those in the existing test set. We found that in the existing training and test sets, all programs contain at least two actions, and also contain at least one `move` action somewhere in the program. These and any undiscovered differences in the distribution of programs seem to have caused the gap in performance.

3.4 Application to Karel: Changing Training Distributions

In Section 3.3, we saw that the existing model performs much more poorly on certain test datasets that we constructed, compared to its performance on the existing test set as reported in Section 3.3.2. In light of the framework in Section 3.2, various imbalances of the salient random variables in the existing training data could have caused these gaps in performance. Then, a natural solution is to train using datasets constructed to avoid undesirable skews in the salient random variables, which should hopefully perform better across a variety of distributions.

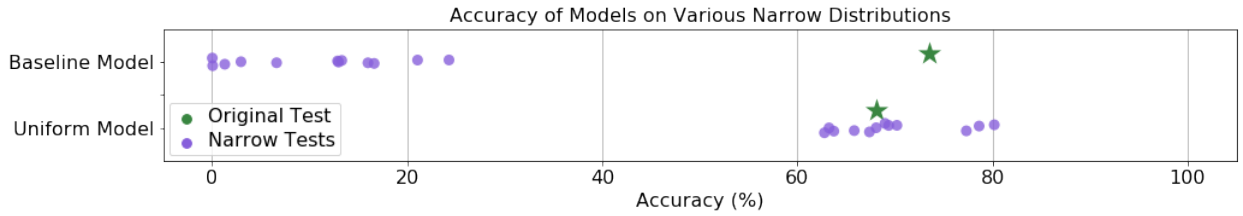


Figure 3.2: Comparison of generalization accuracies across the datasets given in Table 3.1 plus performance on the original test set. The training datasets used for both models (denoted *Baseline* and *Uniform*) contained the same programs; however, for *Uniform*, we sampled new I/O grids used to specify the programs, with homogenized salient random variables, as described in Section 3.3.2.

3.4.1 Training Datasets with Uniform Input-Output Distribution

We generated a training dataset by taking the programs of the existing training set and synthesizing input-output pairs using the procedure described in Section 3.3.2. Furthermore, to make the "number of grids" salient variable uniform, we modify the training procedure by uniformly sampling a number between 1 and 5 for each mini-batch, and using that many input-output pairs to specify the program to the model. We trained a model on this data and then evaluated it on the same set of narrow distribution evaluation datasets as mentioned in Section 3.3.2. Table 3.1 and Figure 3.2 compares how this new model performs to the baseline model. The model trained on uniform input-output distributions maintains much higher generalization accuracy on the test sets of Section 3.3.2 than the baseline model. Note that the uniform input-output distribution is not simply a union of the tested distributions and is intended to cover all possible input specifications.

3.4.2 Real-world Benchmarks

We evaluated both the baseline model, and the uniform model described in the previous paragraph, on a set of 36 real-world Karel programming problems. This dataset was compiled from the Hour of Code Initiative and Stanford University's introductory computer science course, CS106A, with the problems being hand-designed as educational exercises for students. We found that the baseline model got 4 correct (11.1%) while the uniform model got 7 correct (19.4%) when both models were trained with 5 shown input-output pairs, i.e., without making the "number of grids" salient variable uniform. This further demonstrates the uniform model's increased ability to generalize to out-of-distribution datasets, including those which are of interest to humans. Both models' accuracies are still low compared to the performance on the synthetically generated test set.

After further analysis, we believe the models face two challenges on the real-world test set:

1. many of the real-world problems require long programs to solve compared to the synthetic test set,

2. the specifications in the real-world examples always contains fewer than 5 input-output pairs, and often only a single one. However, the original training methodology for the model assumes that it is provided with a diverse set of 5 input-output pairs. When we modified the training procedure to vary the number of shown input-output examples as in Section 3.4.1, the baseline model got 12 correct (33.3%) and the uniform model got 11 correct (30.6%). This shows that the homogenization on the number of input-output pairs was effective. Overall, the real-world dataset's input-output distribution was similar to the existing training dataset in terms of the salient random variables we homogenized, so it is unsurprising that the baseline model was able to outperform the uniform model, consistent with the results on the existing test set in Figure 3.2.

3.4.3 Augmenting Dataset with Action-only Programs

We observed in Section 3.3.3 that the model fails to do well on either action-only programs or programs with many control-flow statements. In the case of action-only programs, we found that the training data had been pruned to only include programs with at least two actions and at least one `move`, and in the case of programs with complex control flow, we found a similar sparsity in the train set.

As discussed in Section 3.2, the principled way to counteract this sparsity is to introduce uniformity into a set of salient variables. This methodology allows us to counteract both naturally sparse data (such as complex control-flow) and spurious data pre-processing (such as enforcing programs to have at least two actions).

In our case, we introduce uniformity into the length of action-only programs by synthesizing 20,000 programs of each length 1 to 20, by uniformly selecting tokens from the five action choices and generating input-output pairs with other salient random variables as close to the original training set as possible; we append these new programs to the original dataset to train a new model. Table 3.2 shows a clear improvement when homogenizing this salient random variable. The new model achieved 71.8% accuracy on the original test set, which is very close to the 73.52% accuracy of the baseline model.

3.4.4 Training Datasets with Narrowly Distributed Input-Output Examples

As done in for the uniform training dataset, we generated "narrow" training datasets by keeping the same programs as in the existing training data and replacing the input-output pairs with the process from Section 3.3.2.

We trained a variety of models and evaluated them on 12 datasets of different input-output feature distributions. Figure 3.3 summarizes the results of evaluating each model on each dataset by noting the performance of the model on the narrow dataset of the same type and the outcome on every other narrow dataset. For the models trained on the low variance datasets, we observed that they all consistently achieved between 60 and 70 percent accuracy on their own training distribution; however, the uniform model was able to achieve similar performance (between 57 and 80 percent accuracy) as shown in Table 3.1 and Figure 3.2. As such, we hypothesize that models

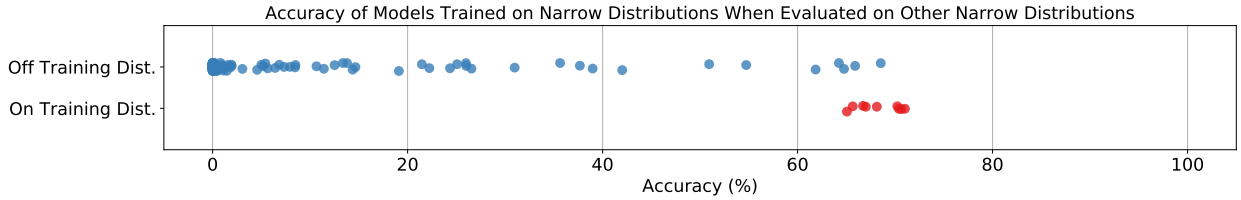


Figure 3.3: Evaluation of models that were trained on narrow distributions. When evaluated on their own training distribution, they consistently achieved similar results. When evaluated on a different narrow distribution, the performance was rarely similar and usually very low.

trained on wide, uniform distributions can still perform comparable to models trained on narrow sub-distributions, even when tested on the same sub-distributions.

3.5 Application to Calculator

The Calculator task is given as follows: given an expression such as `"5+4*(2+3)"`, compute the result modulo 10; in this case, 5. Calculator is a program induction task rather than a program synthesis task like Karel; nevertheless, creating data for the Calculator problem involves sampling from a context-free grammar. Additionally, Calculator is not as intricate a domain as Karel and thus we can more completely control the environment of data generation with less fear of lurking variables.

3.5.1 Calculator Environment

Calculator model. Similar to the work by Zaremba and Sutskever (2015), we implement an LSTM that parses calculator expressions on a character level. We perform a 10-class classification problem using a dense network on the final hidden state of the LSTM. The prediction is correct if it exactly matches the evaluation result of the expression, modulo 10.

Distributions of Calculator tasks. We propose 4 distributions for calculator tasks: direct CFG sampling (`DCFG`), tensor2tensor sampling (`T2T`), "runs" CFG sampling (`RCFG`), and balanced sampling (`BAL`).

Two of our distributions represent reasonable ways in which a researcher might choose to sample data. The first is `DCFG`, which involves returning a digit with some probability $(1 - p)$, or else recursively sampling two productions and combining them with a $-$, $*$, or $+$, each with probability $\frac{p}{3}$. This corresponds to a direct, weighted sampling of the CFG for the calculator grammar. The second is `T2T`, which is used by the tensor2tensor library to sample arithmetic expressions in one

of its examples.[2] It involves sampling a depth d, then ensuring that the resulting AST has depth d by forcing a random side of the operation production to be sampled to $d-1$ and the other side to be sampled to a depth $d' \sim \mathcal{U}\{0, 1, \ldots, d-1\}$.

The other two distributions represent potentially difficult or nonstandard problems that might appear in practical environments. `RCFG` is similar to `DCFG` but involves increasing the frequency of "runs" of the associative operations $+$ and $*$ by picking 2, 3, or 4 subexpressions and then combining them with the given symbol. `BAL` (balanced sampling) involves selecting a depth and then creating an AST that is a balanced binary tree at that depth. Importantly, regardless of sampling technique, redundant parentheses are removed. This is to increase the difficulty somewhat as order of operations needs to be established.

3.5.2 Salient Variables and Methodology

We use the following salient variables: length (rounded to the nearest even number), number of operations, number of pairs of parentheses, mean parenthesized depth, and maximum parenthesized depth. Parenthesized depth is defined for each digit and refers to the number of nested parentheses it is in. For example in `(1+2)*(3-4)+5`, the 1, 2, 3, and 4 are at depth 1 while the 5 is at depth 0.

We constructed $2 \times (1+5)$ distributions in total, corresponding to a total of 2 task distributions, `T2T` and `DCFG`, which represent the "natural" sampling techniques a researcher might employ, and $1+5$ homogenization strategies, one unhomogenized and five homogenized corresponding to each salient variable with $\varepsilon = 0.025$. We then evaluated each model on a fresh evaluation set sampled from a mixture of the four unhomogenized distributions (`T2T`, `DCFG`, `RCFG`, `BAL`).

3.5.3 Results

The original performances and improvements created by homogenizing different random variables can be found in Table 3.3. On average, homogenizing the `DCFG` and `T2T` distributions caused the accuracy to increase by 5.00pp and 2.84pp, respectively.

We note that the Calculator domain is much simpler than Karel when considering both input complexity (grid worlds versus arithmetic expressions) and output complexity (a DSL program versus a single digit). Furthermore, the difference in distributions between the naive sampling approaches and the versions with one homogenized random variable are not as different in Calculator as what we observed in Karel (see Table 3.1 for the dramatic effect of $\mathcal{D}_{\text{marker}}$). We hypothesize that it is this difference in complexity that explains the smaller (but still consistent) effect of homogenizing salient random variables in Calculator as compared to in Karel.

[2] https://github.com/tensorflow/tensor2tensor/blob/8bd81e8fe9dafd4eb1dfa519255bcbe3e33c7ffa/tensor2tensor/data_generators/algorithmic_math.py

Table 3.3: Improvements in Calculator performance over unhomogenized distributions when various homogenizations were applied. See Section 3.5 for details on performance metrics.

	Original	Length	Max Depth	Mean Depth	#Operations	#Parens
T2T	83.83%	+4.35pp	+4.24pp	+2.14pp	+1.19pp	+2.32pp
DCFG	78.25%	+3.84pp	+5.92pp	+4.02pp	+6.72pp	+4.51pp

3.6 Related Work

In certain domains like computer vision and robotics, collecting high-quality real-world training data incurs significant cost, and so many researchers have investigated the use of large amounts of synthetic data. For example, Christiano et al. (2016), Peng et al. (2018), Pinto et al. (2018), and Bousmalis et al. (2018) aim to learn robotics policies that compensate for differences between the real world and the simulation. Within computer vision, Shrivastava et al. (2017) demonstrate learning from entirely synthetic images for gaze and pose estimation.

3.7 Discussion

We demonstrate that existing sampling methods for randomly generating input-output examples have unintended and overlooked distribution flaws in both the Calculator and the Karel domain. These flaws prevent models trained on these distributions from generalizing to other test distributions, even if the are very simple. To resolve these problems, we propose a robust strategy for controlling and evaluating the bias of synthetic data distributions over programs and specifications by defining certain random variables that capture desired features of the program and input spaces, such as the number of parentheses in a calculator expression, and specifically manipulating their distributions. Equipped with our method, deep networks exhibit an increase in cross-distribution test accuracy, at the expense of a minor decrease in on-distribution test accuracy. We believe this methodology would lead to more rigorous evaluation of the synthesis techniques and moreover, aid them in learning better models that generalize well.

Equipped with a set of hand-designed salient random variables, we demonstrate the effectiveness of homogenizing synthetic datasets over this set. One of the core limitations of our approach is that the salient random variables are engineered by hand. This requires the scientist to have insights about the structure of the training examples they are randomly generating. Therefore, a promising extension of this algorithm is to automatically select which salient random variables to use, and automatically compute these variables; potentially via the use of a general unsupervised learning algorithm.

We evaluate our method on two domains: the Karel DSL, and a calculator expression parser. There is a natural question of whether the methods developed in this paper will improve out-of-distribution generalization on applications other than program synthesis which use synthetic data—for example, a convolutional neural network that receives renderings of a virtual environment,

for the robotics and vision domains mentioned in Section 3.6. Providing a thorough evaluation of our proposed homogenization algorithm on alternative domains is a promising area for future work.

Part II

Synthesis from Natural Language

Chapter 4

Synthesis of SQL with Relation-Aware Self-Attention

In the previous two chapters, we addressed issues relating to generalization and embodiment primarily in the Karel domain, where we synthesize imperative programs with control flow from input-output examples. In this chapter[1] and in Chapter 5, we will address challenges relating to generalization and abstraction in program synthesis from natural language, or semantic parsing.

In this chapter, we develop an improved encoder for the input specification to enable better generalization in generating SQL queries from natural language questions. This task, also known as text-to-SQL, is particularly challenging as we would like our methods to work with arbitrary new *database schemas* different from what was available at training time. For example, we might want to answer questions on a database about Olympic athletes even if the training dataset only contained other topics like cars or movies. To handle complex questions for novel database schemas, it is critical to properly encode the schema as part of the input with the question. By using *relation-aware self-attention* within the encoder, the model can better reason about how the tables and columns relate to each other and with the words in the question, allowing it to deliver significantly higher accuracy than previous work. On the challenging Spider dataset (Yu et al. 2018c), which tests for generalization to novel database schemas and query structures, we achieve a 42.94% accuracy, significantly higher than a published baseline of 18.9% Yu et al. (2018b).

4.1 Introduction

The ability to effectively query databases with natural language has the potential to unlock the power of large datasets to the vast majority of users who are not proficient in the use of languages such as SQL. As such, a large body of existing work has focused on the task of translating natural language questions into queries that existing database software can execute.

[1]The material in this chapter is based on Shin (2019), a preprint. A later version of the work was published as Wang et al. (2020).

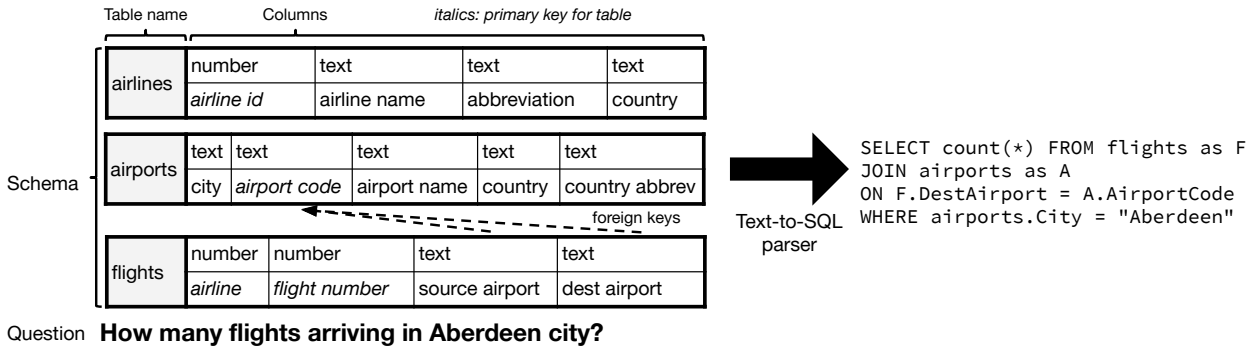


Figure 4.1: Overview of text-to-SQL task. This chapter proposes and evaluates the use of relation-aware self-attention to encode the question and schema, including elements such as the "foreign key" relationship shown.

The release of large annotated datasets containing questions and the corresponding database queries has catalyzed significant progress in the field, by enabling the training of supervised learning models for the task (Zhong et al. 2017; Finegan-Dollak et al. 2018). This progress has arrived not only in the form of improved accuracy on the test sets provided with the datasets, but also through an evolution of the problem formulation towards greater complexity more closely resembling real-world applications.

The recently-released Spider dataset (Yu et al. 2018c) exemplifies greater realism in the task specification: the queries are written using SQL syntax, the dataset contains a large number of domains and schemas with no overlap between the train and test sets, and each schema contains multiple tables with many complicated questions being expressed in the queries. Due to the extra difficulty caused by these factors, the best result on this dataset in published work achieves about 19% exact match accuracy on the development set (Yu et al. 2018b), which is significantly worse compared to $> 80\%$ exact matching accuracy reported for past datasets such as ATIS, GeoQuery, and WikiSQL (Yu et al. 2018c; *A Large Annotated Semantic Parsing Corpus for Developing Natural Language Interfaces.* 2019).

We posit that a central challenge of the multi-schema problem setting is generalization to new database schemas different from what was seen during training. When the model needs to generate queries for arbitrary new schemas, it needs to take the relevant schema as an input and process it together with the question in order to generate the correct query.

Previous methods on the WikiSQL dataset (Zhong et al. 2017) have also contended with the challenge of generalizing to arbitrary new schemas. However, all schemas in this dataset are quite simple, as they only contain one table. The model has no need to reason about the relationships between multiple tables in order to generate the correct query. As such, models developed for this dataset have largely focused on innovations to the decoder for generating the query, rather than the encoder for the question and the schema. In contrast, most real databases (including those in Spider) contain multiple tables with features such as foreign keys that link rows in one table to another.

We hypothesize that to generate correct queries for such databases, a model needs the ability to reason about how the tables and columns in the provided schema relate to each other and use this information in interpreting the question.

We develop a method to test this hypothesis. First, we construct a directed graph (with labels on nodes and edges) over all of the elements of the schema. This graph contains a node for each column or table, and an edge exists from one node to another if the two have an interesting relationship (e.g., the two nodes are columns which belong to the same table) with a label encoding that relationship. Each node has an initial vector representation based on the words in the column or table's name. We also obtain a vector representation for each word in the question. For a fixed number of times, we then update each node and word representation based on all other node and word representations, taking the labels of edges between nodes into account. We use these updated representations with a tree-structured SQL decoder, which uses attention over them at each decoding step, and also points to the column and table representations when it needs to output a column or table reference in the query.

We empirically evaluate our method on the Spider dataset (Yu et al. 2018c), using a decoder based on Yin and Neubig (2017). We achieve 42.94% exact set match accuracy on the development set, significantly higher than a prior published result of 18.9% (Yu et al. 2018b). We further verify the utility of directly encoding the relationships within the schema with an ablation study.

4.2 Problem Formulation and Motivation

Provided with a natural language question and a schema for a relational database, our goal is to generate the SQL query corresponding to the question. The schema contains the following information, as depicted in Figure 4.1: a list of *tables* in the database, each with a meaningful name (e.g., AIRLINES, AIRPORTS, and FLIGHTS for an aviation database); for each table, a list of *columns*, where each column has a type such as `number` or `text`, and some of them can be *primary keys*, used to uniquely identify each row; finally, a column can have another column in a different table as its *foreign key*, which is used to link together rows across multiple tables. As mentioned in the introduction, we would like our method to generalize to not only new questions, but also new schemas it has never seen during training time.

Using natural language to query databases has been a long-standing problem studied for many decades in the research community (Androutsopoulos et al. 1995; Popescu et al. 2004). We identify several limitations of past work and problem settings:

(a) Some datasets only concern themselves with one domain (e.g., US geography (Zelle and Mooney 1996)).

(b) Most datasets about one domain also contain only one database schema for the domain, so the system only needs to know how to generate queries for that single schema.

(c) While WikiSQL (Zhong et al. 2017) contains a large number of domains and schemas, each schema only contains one table in it.

(d) Datasets containing only one domain and database necessarily contain overlaps between the train and test sets. Furthermore, as discussed by Finegan-Dollak et al. (2018), many existing datasets exhibit overlap in queries between the train and test sets, which limits their ability to test how models generalize to generating new queries.

The neural methods common in recent work follow an encoder-decoder paradigm, and past work has largely focused on improvements to the decoder part. As such, the question of how best to encode the question and the schema has remained relatively under-studied. Models developed using datasets which contain only one domain and schema ((a) and (b) above) typically internalize the schema within the learned parameters. The popular WikiSQL dataset necessitates generalizing to new schemas at test time, so models developed for it also encode the schema together with the question; however, as all these schemas only contain one table, the demands placed on the schema encoder are relatively light.

It is most useful if we can train a single model that can generalize to new domains and new database schemas, where both the queries and the schemas have complicated structure that better reflect potential real-world applications. The Spider dataset (Yu et al. 2018c) provides an environment for evaluating this problem setting. In this work, we study how to better encode the question and schema under these more demanding conditions.

4.3 Existing Encoding Schemes

In this section, we review how some existing works (mostly for the WikiSQL dataset) addressed the challenge of encoding the input question and schema.

Encoding each element independently In SQLNet (Xu et al. 2017) (for the WikiSQL dataset), the name of each column, and the question, are separately processed using a bidirectional LSTM. The LSTM outputs for the question tokens are utilized in the decoder using attention, and the final LSTM states of the columns with a pointer network. Note that the encoding of each column is uninfluenced by which other column are present; furthermore, the question is encoded entirely separately from the schema.

In SyntaxSQLNet (Yu et al. 2018b) (for the Spider dataset), the question is encoded identically as SQLNet, using a bidirectional LSTM. Each column is encoded similarly, by using a bidirectional LSTM over the concatenation of the words in the column name, words in the table name, and column type (e.g., `number`, `string`).

Encoding the columns jointly TypeSQL (Yu et al. 2018a) computes the encoding of each column by an elementwise averaging of the embeddings of the words in the name, and using a bidirectional LSTM over these averages (i.e., over all columns); therefore, the encoding for each column depends on which other columns are present (and also their order, although that is arbitrary).

Using the schema while encoding the question Using the information in the schema while encoding the question can help the decoder generate the correct query. In TypeSQL, the word embeddings for each question token are concatenated with a *type* embedding; in particular, question tokens appearing in a column name are specially marked.

Coarse2Fine (Dong and Lapata 2018) goes further by using attention to gather information from the schema while encoding the question. First, the input question is encoded using a bidirectional LSTM, then an attention mechanism retrieves a weighted sum of the column embeddings for the LSTM state of each token. These two are concatenated together and processed together in another bidirectional LSTM, to obtain the final embeddings for each question token.

IncSQL (Shi et al. 2018) uses "cross-serial attention", also updating the column embeddings using the question token embeddings, in addition to the other direction used in Coarse2Fine.

4.4 Our Approach

In the previous section, we reviewed how previous neural methods developed for the text-to-SQL problem encode the input (the question and the database schema) for use in the decoder. Several of these methods encode the question and the columns entirely independently (e.g., the embedding of a column is uninfluenced by other columns in the schema).

In contrast, we specifically seek interactions between schema elements within our encoder, as explained in Sections 4.1 and 4.2. In this section, we describe how we encode the schema as a directed graph and use relation-aware self-attention to interpret it. We will use the following notation:

- c_i for each column in the schema. Each column contains words $c_{i,1}, \cdots, c_{i,|c_i|}$.
- t_i for each table in the schema. Each table contains words $t_{i,1}, \cdots, t_{i,|t_i|}$.
- q for the input question. The question contains words $q_1, \cdots, q_{|q|}$.

4.4.1 Encoding the Schema as a Graph

To support reasoning about relationships between schema elements in the encoder, we begin by representing the database schema using a directed graph $\mathcal{G}$, where each node and edge has a label. We represent each table and column in the schema as a node in this graph, labeled with the words in the name; for columns, we prepend the type of the column to the label. For each pair of nodes x and y in the graph, Table 4.1 describes when there exists an edge from x to y and the label it should have. Figure 4.2 illustrates an example graph (although not all edges and labels are shown).

4.4.2 Initial Encoding of the Input

We now obtain an initial representation for each of the nodes in the graph, as well as for the words in the input question. For the graph nodes, we use a bidirectional LSTM over the words contained

Table 4.1: Description of edge types present in the directed graph created to represent the schema. An edge exists from node x to node y if the pair fulfills one of the descriptions listed in the table, with the corresponding label. Otherwise, no edge exists from x to y.

Type of x	Type of y	Edge label	Description
Column	Column	SAME-TABLE	x and y belong to the same table.
		FOREIGN-KEY-COL-F	x is a foreign key for y.
		FOREIGN-KEY-COL-R	y is a foreign key for x.
Column	Table	PRIMARY-KEY-F	x is the primary key of y.
		BELONGS-TO-F	x is a column of y (but not the primary key).
Table	Column	PRIMARY-KEY-R	y is the primary key of x.
		BELONGS-TO-R	y is a column of x (but not the primary key).
Table	Table	FOREIGN-KEY-TAB-F	Table x has a foreign key column in y.
		FOREIGN-KEY-TAB-R	Same as above, but x and y are reversed.
		FOREIGN-KEY-TAB-B	x and y have foreign keys in both directions.

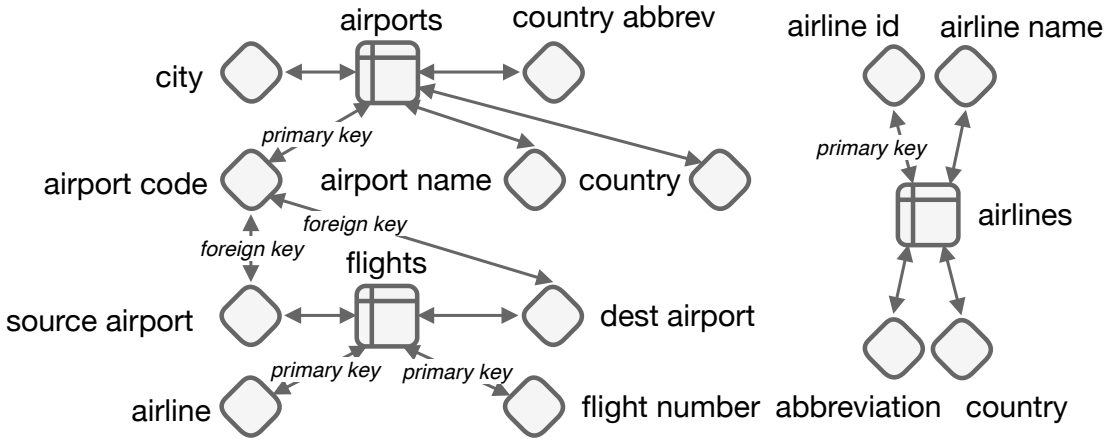
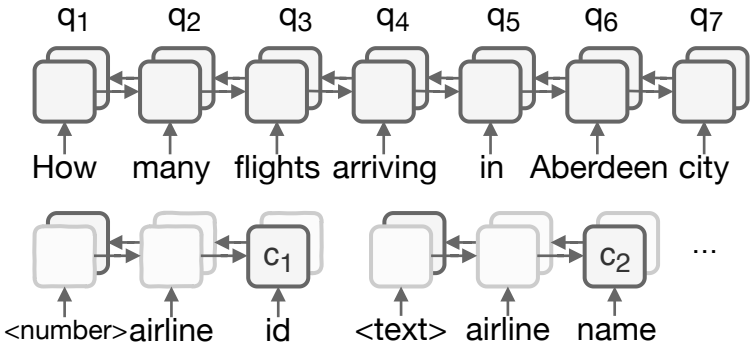
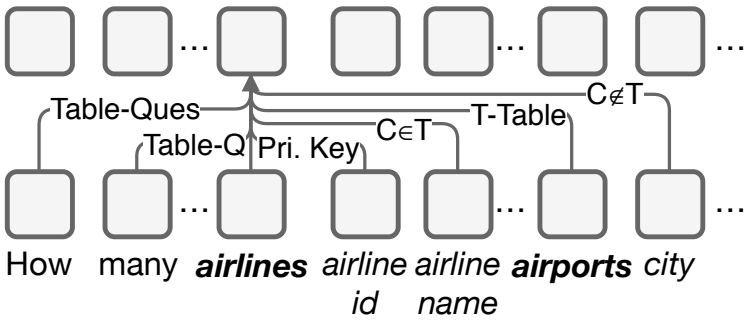


Figure 4.2: An illustration of an example schema as a graph. We do not depict all edges and label types of Table 4.1 to reduce clutter.

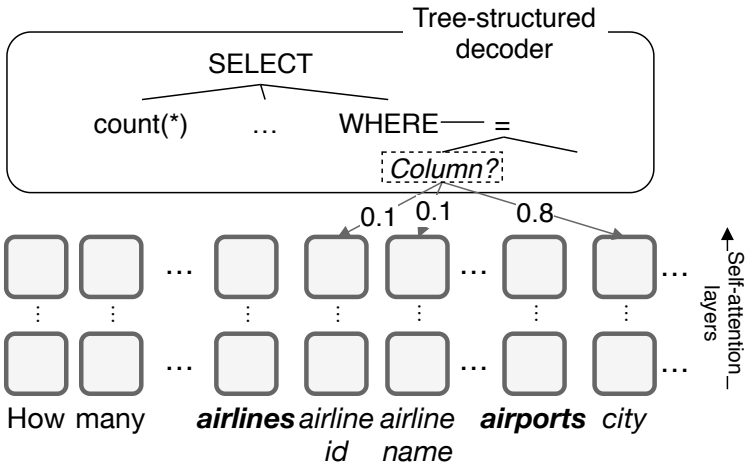
in the label. We concatenate the output of the initial and final time steps of this LSTM to form the embedding for the node. For the question, we also use a bidirectional LSTM over the words.



(a) Initial encoding of the input (Section 4.4.2)



(b) One layer of relation-aware self-attention (Section 4.4.3)



(c) The decoder, choosing a column (Section 4.4.4)

Figure 4.3: Overview of the stages of our approach.

Formally, we perform the following:

$$(\mathbf{c}^{\text{fwd}}_{i,0}, \mathbf{c}^{\text{rev}}_{i,0}), \cdots, (\mathbf{c}^{\text{fwd}}_{i,|c_i|}, \mathbf{c}^{\text{rev}}_{i,|c_i|}) = \text{BiLSTM}_{\text{Column}}(c^{\text{type}}_i, c_{i,1}, \cdots, c_{i,|c_i|}); \quad \mathbf{c}^{\text{init}}_i = \text{Concat}(\mathbf{c}^{\text{fwd}}_{i,|c_i|}, \mathbf{c}^{\text{rev}}_{i,0})$$
$$(\mathbf{t}^{\text{fwd}}_{i,1}, \mathbf{t}^{\text{rev}}_{i,1}), \cdots, (\mathbf{t}^{\text{fwd}}_{i,|t_i|}, \mathbf{t}^{\text{rev}}_{i,|t_i|}) = \text{BiLSTM}_{\text{Table}}(t_{i,1}, \cdots, t_{i,|t_i|}); \quad \mathbf{t}^{\text{init}}_i = \text{Concat}(\mathbf{t}^{\text{fwd}}_{i,|c_i|}, \mathbf{t}^{\text{rev}}_{i,1})$$
$$(\mathbf{q}^{\text{fwd}}_1, \mathbf{q}^{\text{rev}}_1), \cdots, (\mathbf{q}^{\text{fwd}}_{|q|}, \mathbf{q}^{\text{rev}}_{|q|}) = \text{BiLSTM}_{\text{Question}}(q_1, \cdots, q_{|q|}); \quad \mathbf{q}^{\text{init}}_i = \text{Concat}(\mathbf{q}^{\text{fwd}}_i, \mathbf{q}^{\text{rev}}_i)$$

where each of the BiLSTM functions first lookup word embeddings for each of the input tokens. The LSTMs do not share any parameters with each other.

4.4.3 Relation-Aware Self-Attention

At this point, we have representations $\mathbf{c}^{\text{init}}_i$, $\mathbf{t}^{\text{init}}_i$, and $\mathbf{q}^{\text{init}}_i$. Similar to encoders used in some previous papers, these initial representations are independent of each other (uninfluenced by which other columns or tables are present). Now, we would like to imbue these representations with the information in the schema graph. We use a form of self-attention (Vaswani et al. 2017) that is relation-aware (Shaw et al. 2018) to achieve this goal.

In one step of relation-aware self-attention, we begin with an input $\mathbf{x}$ of n elements (where $x_i \in \mathbb{R}^{d_x}$) and transform each x_i into $y_i \in \mathbb{R}^{d_z}$. We follow the formulation described in Shaw et al. (2018):

$$e^{(h)}_{ij} = \frac{x_i W^{(h)}_Q (x_j W^{(h)}_K + \mathbf{r^K_{ij}})^T}{\sqrt{d_z/H}}; \quad \alpha^{(h)}_{ij} = \frac{\exp(e^{(h)}_{ij})}{\sum_{l=1}^{n} \exp(e^{(h)}_{il})}$$
$$z^{(h)}_i = \sum_{j=1}^{n} \alpha^{(h)}_{ij} (x_j W^{(h)}_V + \mathbf{r^V_{ij}}); \quad z_i = \text{Concat}(z^{(0)}_i, \cdots, z^{(H)}_i)$$
$$\tilde{y}_i = \text{LayerNorm}(x_i + z_i); \quad y_i = \text{LayerNorm}(\tilde{y}_i + \text{FC}(\text{ReLU}(\text{FC}(\tilde{y}_i)))$$

The r_{ij} terms encode the relationship between the two elements x_i and x_j in the input. We explain how we obtain r_{ij} in the next part.

Application Within Our Encoder At the start, we construct the input x of $|c| + |t| + |q|$ elements using $\mathbf{c}^{\text{init}}_i$, $\mathbf{t}^{\text{init}}_i$, and $\mathbf{q}^{\text{init}}_i$:

$$x = (\mathbf{c}^{\text{init}}_1, \cdots, \mathbf{c}^{\text{init}}_{|c|}, \mathbf{t}^{\text{init}}_1, \cdots, \mathbf{t}^{\text{init}}_{|t|}, \mathbf{q}^{\text{init}}_1, \cdots, \mathbf{q}^{\text{init}}_{|q|}).$$

We then apply a stack of N relation-aware self-attention layers, where N is a hyperparameter. We set $d_z = d_x$ to facilitate this stacking. The weights of the encoder layers are not tied; each layer has its own set of weights.

We define a discrete set of possible relation types, and map each type to an embedding to obtain r^V_{ij} and r^K_{ij}. We need a value of r_{ij} for every pair of elements in x. If x_i and x_j both correspond to nodes in $\mathcal{G}$ (i.e. each is either a column or table) with an edge from x_i to x_j, then we use the label on that edge (possibilities listed in Table 4.1).

However, this is not sufficient to obtain r_{ij} for every pair of i and j. In the graph we created for the schema, we have no nodes corresponding to the question words; not every pair of nodes in the graph has an edge between them (the graph is not complete); and we have no self-edges (for when $i = j$). As such, we add more types beyond what is defined in Table 4.1:

- If $i = j$, then COLUMN-IDENTITY or TABLE-IDENTITY.
- $x_i \in$ question, $x_j \in$ question: QUESTION-DIST-d, where

$$d = \text{clip}(j - i, D)$$
$$\text{clip}(a, D) = \max(-D, \min(D, a)).$$

  We use $D = 2$.

- $x_i \in$ question, $x_j \in$ column $\cup$ table; or $x_i \in$ column $\cup$ table, $x_j \in$ question: QUESTION-COLUMN, QUESTION-TABLE, COLUMN-QUESTION or TABLE-QUESTION depending on the type of x_i and x_j.
- Otherwise, one of COLUMN-COLUMN, COLUMN-TABLE, TABLE-COLUMN, or TABLE-TABLE.

In the end, we add $2 + 5 + 4 + 4$ types beyond the 10 in Table 4.1, for a total of 25 types.

After processing through the stack of N encoder layers, we obtain

$$(\mathbf{c}_1^{\text{final}}, \cdots, \mathbf{c}_{|c|}^{\text{final}}, \mathbf{t}_1^{\text{final}}, \cdots, \mathbf{t}_{|t|}^{\text{final}}, \mathbf{q}_1^{\text{final}}, \cdots, \mathbf{q}_{|q|}^{\text{final}}) = y.$$

We use $\mathbf{c}_i^{\text{final}}$, $\mathbf{t}_i^{\text{final}}$, and $\mathbf{q}_i^{\text{final}}$ in our decoder.

Comparison to Past Work We use the same formulation of relation-aware self-attention as Shaw et al. (2018). However, that work only applied it to sequences of words in the context of machine translation, and as such, their r_{ij} only encoded the relative distance between two words. We extend their work and show that relation-aware self-attention can effectively encode more complex relationships that exist within an unordered sets of elements (in this case, columns and tables within a database schema).

Compared to the encoders used in past work such as Coarse2Fine (Dong and Lapata 2018) and IncSQL (Shi et al. 2018), our novel use of relation-aware self-attention frees our encoder from spurious consideration of the order in which the columns and tables are presented in the schema (as the relations we have defined are not impacted by this order).

In their implementation, Shaw et al. (2018) share r_{ij}^K across the H heads and the b examples in a batch, which meant they could use n parallel multiplications of $bH \times (d_Z/H)$ and $(d_z/H) \times n$ matrices. This is possible as r_{ij}^K does not change across the batch when only encoding the relative distances between words. However, due to the more varied relations between x_i in our work which are not shared by all elements in a batch, we instead use bn parallel multiplications of $H \times (d_z/H)$ and $(d_z/H) \times n$ matrices, exploiting the fact that we share r_{ij}^K across the H heads.

4.4.4 Decoder

Once we have obtained an encoding of the input, we used the decoder from Yin and Neubig (2017) to generate the SQL query. The decoder generates the SQL query as an abstract syntax tree[2] in depth-first traversal order, by outputting a sequence of *decoder actions* that either expands the last generated node in the tree using a grammar rule, or chooses a column from the schema. The decoder is restricted to choosing only syntactically valid production rules, and therefore it always produces syntactically valid outputs.

Our decoder is based on Yin and Neubig (2017), although we made two notable modifications. First, when the decoder needs to output a column, we use a pointer network based on scaled dot-product attention (Vaswani et al. 2017) which points to $\mathbf{c}_i^{\text{final}}$ and $\mathbf{t}_i^{\text{final}}$. Second, at each step, the decoder accesses the encoder outputs $\mathbf{c}_i^{\text{final}}$, $\mathbf{t}_i^{\text{final}}$, and $\mathbf{q}_i^{\text{final}}$ using multi-head attention. The original decoder in Yin and Neubig (2017) uses a simpler form of attention.

4.5 Experiments

In this section, we describe the experiments we conducted to empirically validate our schema encoding approach.

4.5.1 Experimental Setup

We implemented our model using PyTorch (Paszke et al. 2019). Within the encoder, we use GloVe word embeddings and hold them fixed during training. All word embeddings have dimension 300. The bidirectional LSTMs have hidden size 128 per direction, and use the recurrent dropout method of Gal and Ghahramani (2016) with rate 0.2. Within the relation-aware self-attention layers, we set $d_x = d_z = 256$, $H = 8$, and use dropout with rate 0.1. The position-wise feed-forward network has inner layer dimension 1024. Inside the decoder, we use rule embeddings of size 128, node type embeddings of size 64, and a hidden size of 256 inside the LSTM with dropout rate 0.2 .

We used the Adam optimizer (Kingma and Ba 2015) with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-9}$, which are defaults in PyTorch. During the first $warmup\_steps = max\_steps/20$ steps of training, we linearly increase the learning rate from 0 to 10^{-3}. Afterwards, the learning rate is annealed to 0, with formula $10^{-3}(1 - \frac{step - warmup\_steps}{max\_steps - warmup\_steps})^{-0.5}$. For all parameters, we used the default initialization method in PyTorch. We use a batch size of 50 and train for up to 40,000 steps.

4.5.2 Dataset and Metrics

We use the Spider dataset (Yu et al. 2018c) for all our experiments. As described by Yu et al. (2018c), the training data contains 8,659 examples, including 1,659 examples (questions and queries, with the accompanying schemas) from the Restaurants (Zelle and Mooney 1996; Tang and Mooney

[2]Following SyntaxSQLNet (Yu et al. 2018b), the FROM clause is excluded from the AST for the decoder to predict; rather, it is recovered afterwards with hand-written rules using the columns referred to in the remainder of the query.

Table 4.2: Exact match accuracy of different models on the development set of Spider. The first row is the SyntaxSQLNet (Yu et al. 2018b) baseline; the second row is our method; the remainder are ablations on our method. The columns refer to different subsets of the development set, from Yu et al. (2018c); "All" refers to all 1,034 dev examples.

Model	Easy	Medium	Hard	Extra Hard	All
SyntaxSQLNet	38.40%	15.00%	16.09%	3.53%	18.96%
Our method	57.20%	44.55%	39.66%	21.18%	**42.94**%
No self-attention layers	42.00%	25.68%	22.99%	5.88%	25.92%
2 self-attention layers	56.00%	45.00%	40.23%	19.41%	42.65%
Fewer relation types	47.60%	30.45%	25.86%	10.00%	30.46%
No relation types	46.80%	29.55%	29.89%	8.82%	30.37%
No pretrained word embeddings	40.80%	29.09%	27.01%	5.88%	27.76%

2000), GeoQuery (Zelle and Mooney 1996), Scholar (Iyer et al. 2017), Academic (Li and Jagadish 2014), Yelp and IMDB (Yaghmazadeh et al. 2017) datasets. We do **not** use the data augmentation scheme of Yu et al. (2018b).

As Yu et al. (2018c) have kept the test set secret, we perform all evaluations using the publicly available development set. There are 1,034 examples in the development set, containing schemas distinct from those in the training set. We report results using the same metrics as Yu et al. (2018b): exact match accuracy on all development set examples, as well as after division into four levels of difficulty. As in previous work, these metrics do not measure the model's performance on generating values within the queries. We report results from the snapshot that obtained the best exact match accuracy across 3 repetitions of each configuration, except for the SyntaxSQLNet baseline where we reuse the pretrained model from the authors.

In addition to results on all of the development set, we also report results on subsets (Easy, Medium, Hard, and Extra Hard) partitioned by complexity of the query as defined by Yu et al. (2018c). These partitions make up 24.18%, 42.55%, 16.83%, and 16.44% respectively.

4.5.3 Variants Tested

Our main result uses the encoder and decoder described previously, with the number N of relation-aware self-attention layers in the encoder set to 4. To further study the utility of our scheme, we also tried the following variations, listed in Table 4.2.

Reducing the number of self-attention layers. Set $N = 0$ and $N = 2$. With $N = 0$, there are no relation-aware self-attention layers; we set $\mathbf{c}_i^{\text{final}} = \mathbf{c}_i^{\text{init}}$, $\mathbf{t}_i^{\text{final}} = \mathbf{t}_i^{\text{init}}$, and $\mathbf{q}_i^{\text{final}} = \mathbf{q}_i^{\text{init}}$. As such, the question words, the words in each column's name, and the words in each table's name are encoded separately using bidirectional LSTMs.

Removing the relation information from the encoder. We would like to measure the impact of providing to the encoder the 25 relation types we defined earlier. In particular, we want to see whether the self-attention mechanism is sufficient within the encoder to obtain a representation for each schema element that is aware of all of the other schema elements, even if we don't explicitly provide information about how the elements are related.

For "fewer relation types", we exclude all of the types in Table 4.1, resulting in 15 rather than 25 possible types. For "minimal relation types", we also merge all of {QUESTION,COLUMN,TABLE}-{QUESTION,COLUMN,TABLE} relations into one, as well as {COLUMN,TABLE}-IDENTITY with QUESTION-DIST-0, and so we only have 6 types.

Not using pretrained word embeddings. The Spider dataset only contains 8,659 training examples, which is significantly smaller than many other datasets used in natural language processing. Furthermore, there is also reduced overlap in the vocabulary between the training and validation/test sets, as they contain different database schemas and domains. Therefore, we measure the impact of using word embeddings learned from only this dataset. We construct a vocabulary consisting of all of the words in the columns, tables, and questions that occur at least 3 times in the training data (the words which occur in columns and tables are counted every time the corresponding schema is used by a question in the data); for each, we randomly initialize an embedding of dimension 300.

4.6 Results and Discussion

Table 4.2 presents our exact match accuracy results on the development set of Spider. For the SyntaxSQLNet row, we obtained the results by running the pretrained model without data augmentation from https://github.com/taoyds/syntaxSQL. Our method exceeds the performance of all other configurations tried, including all ablations. In particular, we can see that our method strongly outperforms SyntaxSQLNet (Yu et al. 2018b), achieving 42.94% exact match accuracy over the 18.96% of the previous work.

Reducing the number of self-attention layers. We can see that the process of relation-aware self-attention is critical for the performance of this encoder, as the accuracy drops precipitously when the self-attention layers are removed. In particular, "no self-attention layers" uses an encoder very similar to SyntaxSQLNet's. We observe fairly marginal gains by using 4 such layers (in "Our method") as opposed to 2 ("2 self-attention layers").

Removing the relation information from the encoder. Comparing against the rows of "No self-attention layers" and "Our method", we see that while having self-attention layers helps increase performance, it is the relation information provided to the encoder that is responsible for most of the gains. The use of self-attention, on its own, contributes relatively little.

Not using pretrained word embeddings. Given the small size of the training data, we confirm that using pretrained word embeddings helps significantly. When we evaluate "No pretrained

word embeddings" on the subset of the development set where all question words have a learned embedding (i.e. no UNKs in the question; 239 out of 1034 examples), then the exact match accuracy recovers to 40.17%; "Our method" achieves 50.21% on this subset, substantialy reducing the gap in accuracy.

4.7 Related Work

As mentioned earlier, RAT-SQL (Wang et al. 2020) builds upon the work in this chapter. That paper adds multiple enhancements to the relation-aware self-attention method described here, to achieve the best result on the leaderboard for the Spider dataset (*Spider: Yale Semantic Parsing and Text-to-SQL Challenge* 2019) as of July 2020. The enhancements include:

- Name-based linking: adding new relation types, between question words $\mathbf{q}_i$ and columns $\mathbf{c}_i$/tables $\mathbf{t}_i$, to explicitly indicate to the encoder when a word in the question occurs in the name of a column or table.
- Value-based linking: similar to name-based linking for columns, but occurs instead when a question word occurs as a value in the column.
- Using BERT (Devlin et al. 2017a) rather than GloVe as the initial encoding, and fine-tuning
- Increasing the number of relation-aware self-attention layers to 8 from 4.
- Adding a question-schema alignment matrix, so that when the decoder needs to generate a column, it first points to a word in the question, which then points to a column. This does not have a large effect on the accuracy, but improves the interpretability of the model.

Chapter 5

Synthesis with Learned Code Idioms

In this chapter,[1] we address the challenge of reasoning about high-level patterns in the target program and low-level implementation details at the same time. We present PATOIS, a system that allows a neural program synthesizer to explicitly interleave high-level and low-level reasoning at every generation step. PATOIS accomplishes this by automatically mining common *code idioms* from a given corpus, incorporating them into the underlying language for neural synthesis, and training a tree-based neural synthesizer to see these idioms during code generation. We evaluate PATOIS on two complex semantic parsing datasets: Spider which we also used in Chapter 4, and Hearthstone Ling et al. 2016, a dataset of Python implementations of cards in a video game. We show that using learned code idioms improves the synthesizer's accuracy.

5.1 Introduction

In the last decade, program synthesis has advanced dramatically thanks to the novel neural and neuro-symbolic techniques (Devlin et al. 2017b; Balog et al. 2016; Kalyan et al. 2018), first mass-market applications (Polozov and Gulwani 2015), and massive datasets (Devlin et al. 2017a; Yin et al. 2018a; Yu et al. 2018c). A particular focus has been on program synthesis from natural language, and Table 5.1 shows a few examples of typical tasks. Most of the successful applications apply program synthesis to manually crafted domain-specific languages (DSLs) such as FlashFill and Karel, or to subsets of general-purpose functional languages such as SQL and Lisp. However, scaling program synthesis to real-life programs in a general-purpose language with complex control flow remains a challenge.

We conjecture that one of the main current challenges of synthesizing a program is insufficient separation between high-level and low-level reasoning. In a typical program generation process, be it a neural model or a symbolic search, the program is generated in terms of its *syntax tokens*, which represent low-level implementation details of the latent high-level *patterns* in the program. In contrast, humans switch between high-level reasoning (*"a binary search over an array"*) and low-level implementation ("`while l < r: m = (l+r)/2 ...`") repeatedly when writing a single

[1] The material in this chapter is based on Shin et al. (2019).

Table 5.1: Representative program synthesis tasks from real-world semantic parsing datasets.

Dataset	Natural Language Specification	Program

Hearthstone (Ling et al. 2016)	*Mana Wyrm (1, 3, 1, Minion, Mage, Common) Whenever you cast a spell, gain +1 Attack.*	`#...`
		`def create_minion(self, player):`
		`  return Minion(1, 3, effects=[`
		`  Effect(SpellCast(), ActionTag(`
		`  Give(ChangeAttack(1)),`
		`  SelfSelector()))])`

Spider (Yu et al. 2018c)	*For each stadium, how many concerts are there?*
	Schema:
	`stadium = {stadium_id, name, ...}, ...`
		`SELECT T2.name, COUNT(*)`
		`FROM concert AS T1 JOIN stadium AS T2`
		`  ON T1.stadium_id = T2.stadium_id`
		`GROUP BY T1.stadium_id`

function. Reasoning over multiple abstraction levels at once complicates the generation task for a model.

This conjecture is supported by two key observations. First, recent work (Dong and Lapata 2018; Murali et al. 2017) has achieved great results by splitting the synthesis process into *sketch generation* and *sketch completion*. The first stage generates a high-level sketch of the target program, and the second stage fills in missing details in the sketch. Such separation improves the accuracy of synthesis as compared to an equivalent end-to-end generation. However, it allows only one stage of high-level reasoning at the root level of the program, whereas **(a)** real-life programs involve common patterns at all syntactic levels, and **(b)** programmers often interleave high-level and low-level reasoning during implementation.

Second, many successful applications of inductive program synthesis such as FlashFill (Gulwani 2011) rely on a manually designed DSL to make the underlying search process scalable. Such DSLs include high-level operators that implement common subroutines in a given domain. Thus, they **(i)** compress the search space, ensuring that every syntactically valid DSL program expresses some useful task, and **(ii)** enable logical reasoning over the domain-specific operator semantics, making the search efficient. However, DSL design is laborious and requires domain expertise. Recently, Ellis et al. (2018a) showed that such DSLs are learnable in the classic domains of inductive program synthesis; in this work, we target general-purpose code generation, where DSL design is difficult even for experts.

In this chapter, we present a system, called PATOIS, that equips a program synthesizer with automatically learned high-level *code idioms* (*i.e.* common program fragments) and trains it to use these idioms in program generation. While syntactic by definition, code idioms often represent useful semantic concepts. Moreover, they *compress* and *abstract* the programs by explicitly representing common patterns with unique tokens, thus simplifying generative process for the synthesis model.

PATOIS has three main components, illustrated in Figure 5.1. First, it employs nonparameteric Bayesian inference to mine the code idioms that frequently occur in a given corpus. Second, it marks the occurrences of these idioms in the training dataset as new named operators in an extended grammar. Finally, it trains a neural generative model to optionally emit these named idioms instead of the original code fragments, which allows it to learn idiom usage conditioned

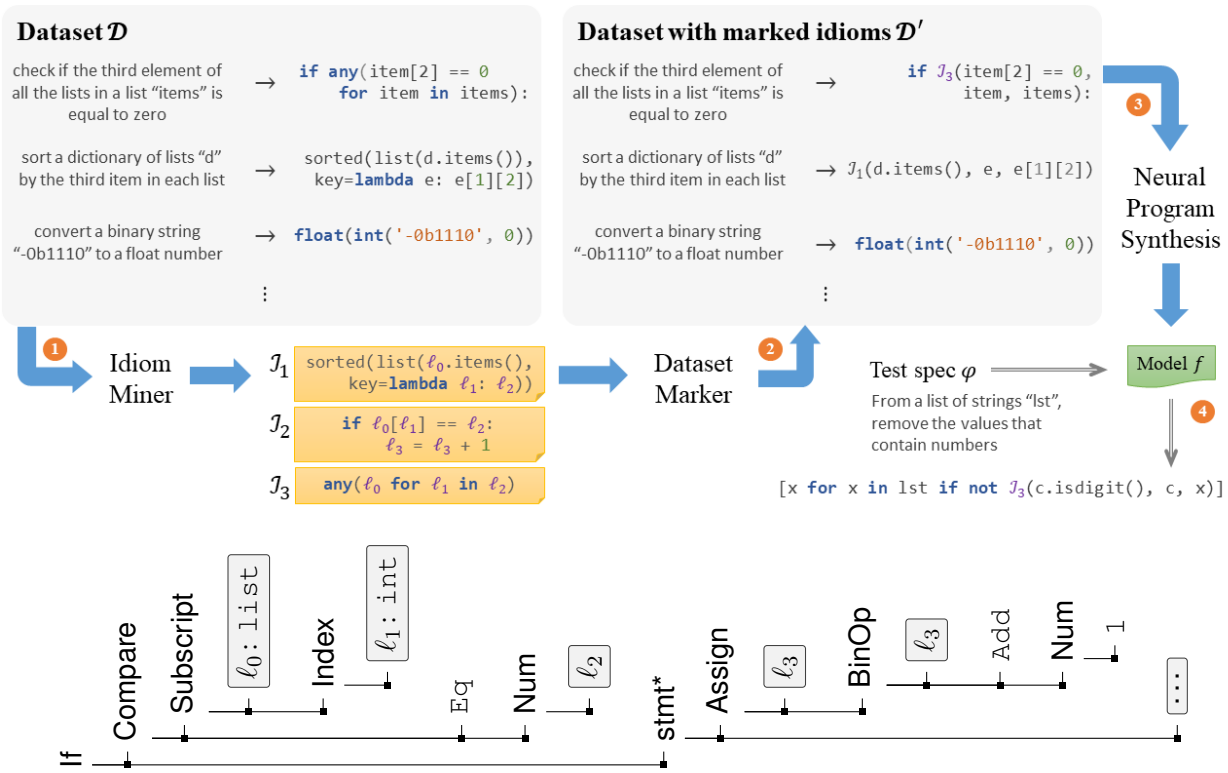


Figure 5.1: *Top:* An overview of PATOIS. A miner ① extracts common idioms from the programs in a given dataset. All the idiom occurrences in the dataset programs are ② marked as optional alternative grammar operators. The dataset with marked occurrences is used to ③ train a neural generative model. At inference time, the model ④ generates programs with named idioms, which are inlined before program execution. Note that idioms may have named subexpressions, may repeat, and may occur at any program level. For clarity, we typeset idioms using function-like syntax $\mathcal{I}_j(\ell_1, \ldots, \ell_k)$ in this chapter, although they are actually represented as AST fragments with no syntax.
Bottom: AST fragment representation of the idiom $\mathcal{I}_2$ in Python. Here sans-serif nodes are fixed non-terminals, `monospaced` nodes are fixed terminals, and boxed nodes are named arguments.

on a task specification. During generation, the model has the ability to emit entire idioms in a single step instead of multiple steps of program tree nodes comprising the idioms' definitions. As a result, PATOIS interleaves high-level idioms with low-level tokens at all levels of program synthesis, generalizing beyond fixed top-level sketch generation.

We evaluate PATOIS on two challenging semantic parsing datasets: Hearthstone (Ling et al. 2016), a dataset of small domain-specific Python programs, and Spider (Yu et al. 2018c), a large dataset of SQL queries over various databases. We find that equipping the synthesizer with learned idioms improves its accuracy in generating programs that satisfy the task description.

5.2 Background

Program Synthesis We consider the following formulation of the *program synthesis* problem. Assume an underlying programming language $\mathcal{L}$ of programs. Each program $P \in \mathcal{L}$ can be represented either as a sequence $y_1 \cdots y_{|P|}$ of its *tokens*, or, equivalently, as an *abstract syntax tree (AST)* T parsed according to the context-free grammar (CFG) $\mathcal{G}$ of the language $\mathcal{L}$. The goal of a program synthesis model $f : \varphi \mapsto P$ is to generate a program P that maximizes the conditional probability $\Pr(P \mid \varphi)$ *i.e.* the most likely program given the specification. We also assume a training set $\mathcal{D} = \{\langle \varphi_j, P_j \rangle\}_{j=1}^{|\mathcal{D}|}$, sampled from an unknown true distribution $\mathfrak{D}$, from which we wish to estimate the conditional probability $\Pr(P \mid \varphi)$.

In this work, we consider general-purpose programming languages $\mathcal{L}$ with a known context-free grammar $\mathcal{G}$ such as Python and SQL. Each *specification* φ is represented as a *natural language task description*, *i.e.* a sequence of words $X = x_1 \cdots x_{|X|}$ (although the PATOIS synthesizer can be conditioned on any other type of incomplete spec). In principle, we do not impose any restrictions on the generative model f apart from it being able to emit syntactically valid programs. However, as we detail in Section 5.4, the PATOIS framework is most easily implemented on top of *structural generative models* such as sequence-to-tree models (Yin and Neubig 2017) and graph neural networks (Li et al. 2016; Brockschmidt et al. 2018).

Code Idioms Following Allamanis and Sutton (2014), we define code idioms as *fragments* $\mathcal{I}$ of valid ASTs T in the CFG $\mathcal{G}$, *i.e.* trees of nonterminals and terminals from $\mathcal{G}$ that may occur as subtrees of valid parse trees from $\mathcal{G}$. The grammar $\mathcal{G}$ extended with a set of idiom fragments forms a *tree substitution grammar* (TSG). We also associate a non-unique *label* ℓ with each nonterminal leaf in every idiom, and require that every instantiation of an idiom $\mathcal{I}$ must have its identically-labeled nonterminals instantiated to identical subtrees. This enables the role of idioms as *subroutines*, where labels act as "named arguments" in the "body" of an idiom. See Figure 5.1 for an example.

5.3 Mining Code Idioms

The first step of PATOIS is obtaining a set of frequent and useful AST fragments as code idioms. The trade-off between frequency and usefulness is crucial: it is trivial to mine *commonly occurring* short patterns, but they are often meaningless (Aggarwal and Han 2014). Instead, we employ and extend the methodology of Allamanis et al. (2018) and frame idiom mining as a nonparameteric Bayesian problem.

We represent idiom mining as inference over *probabilistic tree substitution grammars* (pTSG). A pTSG is a probabilistic context-free grammar extended with production rules that expand to a whole AST fragment instead of a single level of symbols (Cohn et al. 2010; Post and Gildea 2009). The grammar $\mathcal{G}$ of our original language $\mathcal{L}$ induces a pTSG $\mathcal{G}_0$ with no fragment rules and with choice probabilities estimated from the corpus $\mathcal{D}$. To construct a pTSG corresponding to the extension of $\mathcal{L}$ with common tree fragments representing idioms, we define a distribution $\mathfrak{G}$ over pTSGs as follows.

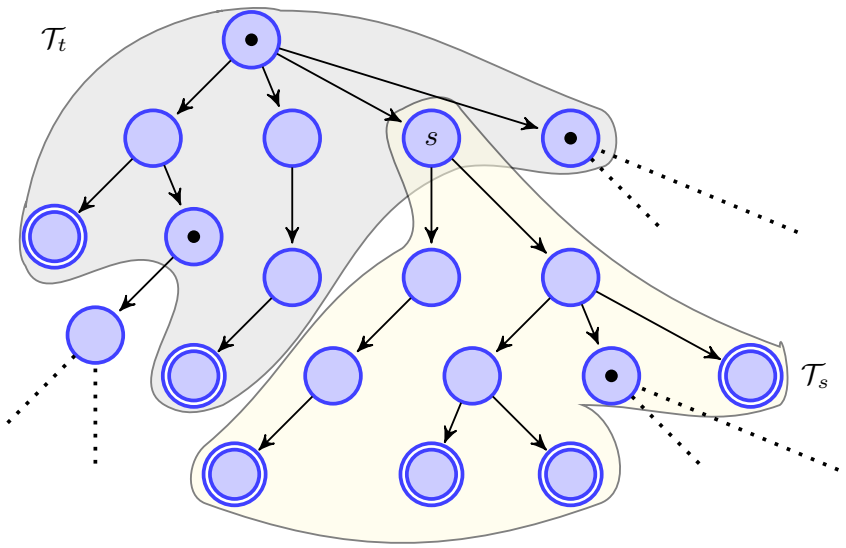


Figure 5.2: MCMC sampling for an AST (figure from (Allamanis and Sutton 2014)). Dots show the inferred nodes where the AST is split into fragments.

We first choose a Pitman-Yor process (Teh and Jordan 2010) as a prior distribution $\mathfrak{G}_0$ over pTSGs. It is a nonparameteric process that has proven to be effective for mining code idioms in prior work thanks to its modeling of production choices as a Zipfian distribution (in other words, it implements the desired "rich get richer" effect, which encourages a smaller number of larger *and* more common idioms). Formally, it is a "stick-breaking" process (Sethuraman 1994) that defines $\mathfrak{G}_0$ as a distribution for each *set of idioms* $\widetilde{\mathcal{I}}_N$ *rooted at a nonterminal symbol* N as

$$\Pr(\mathcal{I} \in \widetilde{\mathcal{I}}_N) \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \pi_k \, \delta\left(\mathcal{I} = \mathcal{I}_k\right), \quad \mathcal{I}_k \sim \mathcal{G}_0$$

$$\pi_k \stackrel{\text{def}}{=} u_k \prod_{j=1}^{k-1} (1 - u_j), \; u_k \sim \text{Beta}\left(1 - d, \, \alpha + kd\right)$$

where $\delta(\cdot)$ is the delta function, and α, d are hyperparameters. See Allamanis et al. (2018) for details.

PATOIS uses $\mathfrak{G}_0$ to compute a posterior distribution $\mathfrak{G}_1 = \Pr\left(\mathcal{G}_1 \mid T_1, \ldots, T_N\right)$ using Bayes' rule, where $T_1, \ldots, T_N$ are concrete AST fragments in the training set $\mathcal{D}$. As this calculation is computationally intractable, we approximate it using type-based MCMC (Liang et al. 2010). At each iteration t of the MCMC process, PATOIS generates a pTSG $\mathcal{G}_t$ whose distribution approaches $\mathfrak{G}_1$ as $t \to \infty$. It works by sampling *splitting points* for each AST T in the corpus $\mathcal{D}$, which by construction define a set of fragments constituting $\mathcal{G}_t$ (see Figure 5.2). The split probabilities of this Gibbs sampling are set in a way that incentivizes merging adjacent tree fragments that often

co-occur in $\mathcal{D}$. The final idioms are then extracted from the pTSG obtained at the last MCMC iteration.

While the Pitman-Yor process helps avoid overfitting the idioms to $\mathcal{D}$, not all sampled idioms are useful for synthesis. Thus we *rank* and *filter* the idioms before using them in the training. In this work, we reuse two ranking functions defined by Allamanis et al. (2018):

$$\mathsf{Score}_{\mathsf{Cov}}(\mathcal{I}) \stackrel{\text{def}}{=} \text{coverage} = \mathsf{count}(T \in \mathcal{D} \mid \mathcal{I} \in T)$$

$$\mathsf{Score}_{\mathsf{CXE}}(\mathcal{I}) \stackrel{\text{def}}{=} \text{coverage} \cdot \text{cross-entropy gain} = \frac{\mathsf{count}(T \in \mathcal{D} \mid \mathcal{I} \in T)}{|\mathcal{D}|} \cdot \frac{1}{|\mathcal{I}|} \log \frac{\Pr_{\mathfrak{G}_1}(\mathcal{I})}{\Pr_{\mathfrak{G}_0}(\mathcal{I})}$$

and also filter out any *terminal* idioms (*i.e.* those that do not contain any named arguments ℓ).

We conclude with a brief analysis of computational complexity of idiom mining. Every iteration of the MCMC sampling traverses the entire dataset $\mathcal{D}$ once to sample the random variables that define the splitting points in each AST. When run for M iterations, the complexity of idiom mining is $\mathcal{O}(M \cdot \sum_{T \in \mathcal{D}} |T|)$. Idiom ranking adds an additional step with complexity $\mathcal{O}(|\widetilde{\mathcal{I}}| \log |\widetilde{\mathcal{I}}|)$ where $\widetilde{\mathcal{I}}$ is the set of idioms obtained at the last iteration. In our experiments (detailed in Section 5.5) we set $M = 10$, and the entire idiom mining takes less than 10 minutes on a dataset of $|\mathcal{D}| \approx 10{,}000$ ASTs.

5.4 Using Idioms in Program Synthesis

Given a set of common idioms $\widetilde{\mathcal{I}} = \{\mathcal{I}_1, \ldots, \mathcal{I}_N\}$ mined by PATOIS, we now aim to learn a synthesis model f that emits whole idioms $\mathcal{I}_j$ as atomic actions instead of individual AST nodes that comprise $\mathcal{I}_j$. Achieving this involves two key challenges.

First, since idioms are represented as AST fragments without concrete syntax, PATOIS works best when the synthesis model f is *structural*, *i.e.* it generates the program AST instead of its syntax. Prior work (Yin and Neubig 2017; Yin et al. 2018b; Brockschmidt et al. 2018) also showed that tree- and graph-based code generation models outperform sequence-to-sequence models, and thus we adopt a similar architecture in this work.

Second, exposing the model f to idiom usage patterns is not obvious. One approach could be to extend the grammar with new named operators $\mathsf{op}_{\mathcal{I}}(\ell_1, \ldots, \ell_k)$ for each idiom $\mathcal{I}$, replace every occurrence of $\mathcal{I}$ with $\mathsf{op}_{\mathcal{I}}$ in the data, and train the synthesizer on the rewritten dataset. However, this would not allow f to learn from the idiom definitions (bodies). In addition, idiom occurrences often overlap, and any deterministic rewriting strategy would arbitrarily discard some occurrences from the corpus, thus limiting the model's exposure to idiom usage. In our experiments, we found that greedy rewriting discarded as many as 75% potential idiom occurrences from the dataset. Therefore, a successful training strategy must preserve all occurrences and instead let the model *learn* a rewriting strategy that optimizes end-to-end synthesis accuracy.

To this end, we present a novel training setup for code generation that encourages the model to choose the most useful subset of idioms and the best representation of each program in terms of the idioms. It works by **(a)** marking occurrences of the idioms $\widetilde{\mathcal{I}}$ in the training set $\mathcal{D}$, **(b)** at

training time, encouraging the model to emit *either* the whole idiom *or* its body for every potential idiom occurrence in the AST, and **(c)** at inference time, replacing the model's state after emitting an idiom $\mathcal{I}$ with the state the model would have if it had emitted $\mathcal{I}$'s body step by step.

5.4.1 Model Architecture

The synthesis model f of PATOIS combines a *spec encoder* f_{enc} and an *AST decoder* f_{dec}, following the formulation of Yin and Neubig (2017). The encoder f_{enc} embeds the NL specification $X = x_1 \cdots x_n$ into word representations $\hat{X} = \hat{\boldsymbol{x}}_1 \cdots \hat{\boldsymbol{x}}_n$. The decoder f_{dec} uses an LSTM to model the sequential generation of the AST in the depth-first order, wherein each timestep t corresponds to *an action* a_t — either (a) expanding a production from the grammar, (b) expanding an idiom, or (c) generating a terminal token. Thus, the probability of generating an AST T given $\hat{X}$ is

$$\Pr(T \mid \hat{X}) = \prod_t \Pr\big(a_t \mid T_t, \hat{X}\big) \tag{5.1}$$

where a_t is the action taken at timestep t, and T_t is the partial AST generated before t. The probability $\Pr(a_t \mid T_t, \hat{X})$ is computed from the decoder's hidden state $\boldsymbol{h}_{t-1}$ depending on a_t.

Production Actions For actions $a_t = \textsc{ApplyRule}[R]$ corresponding to expanding production rules $R \in \mathcal{G}$ from the original CFG $\mathcal{G}$, we compute the probability $\Pr(a_t \mid T_t, \hat{X})$ by encoding the current partial AST structure similarly to Yin and Neubig (2017). Specifically, we compute the new hidden state as $\boldsymbol{h}_t = f_{\text{LSTM}}\left([\boldsymbol{a}_{t-1} \parallel \boldsymbol{c}_t \parallel \boldsymbol{h}_{p_t} \parallel \boldsymbol{a}_{p_t} \parallel \boldsymbol{n}_{f_t}],\ \boldsymbol{h}_{t-1}\right)$ where $\boldsymbol{a}_{t-1}$ is the embedding of the previous action, $\boldsymbol{c}_t$ is the result of soft attention applied to the spec embeddings $\hat{X}$ as per Bahdanau et al. (2015), p_t is the timestep corresponding to expanding the parent AST node of the current node, and $\boldsymbol{n}_{f_t}$ is the embedding of the current node type. The hidden state $\boldsymbol{h}_t$ is then used to compute probabilities of the syntactically appropriate production rules $R \in \mathcal{G}$:

$$\Pr(a_t = \textsc{ApplyRule}[R] \mid T_t, \hat{X}) = \mathsf{softmax}_R\left(g(\boldsymbol{h}_t)\right) \tag{5.2}$$

where $g(\cdot)$ is a 2-layer MLP with a $\tanh$ non-linearity.

Terminal Actions For actions $a_t = \textsc{GetToken}[y]$, we compute the probability $\Pr(a_t \mid T_t, \hat{X})$ by combining a small vocabulary $\mathcal{V}$ of tokens commonly observed in the training data with a *copying mechanism* (Ling et al. 2016; See et al. 2017) over the input X to handle UNK tokens. Specifically, we learn two functions $p_{\text{gen}}(\boldsymbol{h}_t)$ and $p_{\text{copy}}(\boldsymbol{h}_t, X)$ such that p_{gen} produces a score for each vocabulary token $y \in \mathcal{V}$ and p_{copy} computes a score for copying the token y from the input. The scores are then normalized across the entries corresponding to the same constant, as in (Yin and Neubig 2017; Brockschmidt et al. 2018).

5.4.2 Training to Emit Idioms

As discussed earlier, training the model to emit idioms presents computational and learning challenges. Ideally, we would like to extend Eq. (5.1) to maximize

$$\mathcal{J} = \sum_{\tau \in \mathcal{T}} \prod_{i=1}^{|\tau|} \Pr(a_{\tau_i} \mid T_{\tau_i}, \hat{X}) \tag{5.3}$$

where $\mathcal{T}$ is a set of different *action traces* that may produce the output AST T. The traces $\tau \in \mathcal{T}$ differ only in their possible choices of idiom actions $\textsc{ApplyRule}[\mathsf{op}_{\mathcal{I}}]$ that emit some tree fragments of T in a single step. However, computing Eq. (5.3) is intractable because idiom occurrences overlap and cause combinatorial explosion in the number of traces $\mathcal{T}$. Instead, we apply Jensen's inequality and maximize a lower bound:

$$\log \mathcal{J} = \log \sum_{\tau \in \mathcal{T}} \prod_{i=1}^{|\tau|} \Pr(a_{\tau_i} \mid T_{\tau_i}, \hat{X}) \geq \log(|\mathcal{T}|) + \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \sum_{i=1}^{|\tau|} \log \Pr(a_{\tau_i} \mid T_{\tau_i}, \hat{X}) \tag{5.4}$$

Let $A(T_t) = \{a_t^*\} \cup I(T_t)$ be the set of all valid actions to expand the AST T_t at timestep t. Here a_t^* is the action from the *original action trace* that generates T using the original CFG and $I(T_t)$ is the set of idiom actions $\textsc{ApplyRule}[\mathsf{op}_{\mathcal{I}}]$ also applicable at the node to be expanded in T_t. Let $c(\mathcal{T}, t)$ also denote the number of traces $\tau \in \mathcal{T}$ that admit an action choice for the AST T_t from the original action trace. Since each action $a \in A(T_t)$ occurs in the sum in Eq. (5.4) with probability $c(\mathcal{T}, t) \,/\, |A(T_t)|$, we can rearrange this sum over traces as a sum over timesteps of the original trace:

$$\begin{aligned}
&\frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \sum_{i=1}^{|\tau|} \log \Pr(a_{\tau_i} \mid T_{\tau_i}, \hat{X}) = \frac{1}{|\mathcal{T}|} \sum_t \sum_{a \in A(T_t)} \frac{c(\mathcal{T}, t)}{|A(T_t)|} \log \Pr(a \mid T_{\tau_i}, \hat{X}) \\
&= \sum_t \frac{1}{|A(T_t)|} \sum_{a \in A(T_t)} \frac{c(\mathcal{T}, t)}{|\mathcal{T}|} \log \Pr(a \mid T_{\tau_i}, \hat{X}) = \mathop{\mathbb{E}}_{T_t \sim \mathcal{T}} \frac{1}{|A(T_t)|} \sum_{a \in A(T_t)} \log \Pr(a \mid T_{\tau_i}, \hat{X}) \\
&\approx \sum_t \frac{1}{|A(T_t)|} \Big[\log \Pr(a_t^* \mid T_t, \hat{X}) + \sum_{\mathcal{I} \in M(T_t)} \log \Pr(a_t = \textsc{ApplyRule}[\mathsf{op}_{\mathcal{I}}] \mid T_t, \hat{X}) \Big]
\end{aligned} \tag{5.5}$$

In the last step of Equation (5.5), we approximate the expectation over ASTs randomly drawn from all traces $\mathcal{T}$ using only the original trace (containing all possible T_t) as a Monte Carlo estimate.

Intuitively, at each timestep during training we encourage the model to emit *either* the original AST action for this timestep *or* any applicable idiom that matches the AST at this step, with no penalty to either choice. However, to avoid the combinatorial explosion, we only teacher-force the original generation trace (*not* the idiom bodies), thus optimizing the bound in Eq. (5.5). Figure 5.3 illustrates this optimization process on an example.

At inference time, whenever the model emits an $\textsc{ApplyRule}[\mathsf{op}_{\mathcal{I}}]$ action, we teacher-force the body of $\mathcal{I}$ by substituting the embedding of the previous action $\boldsymbol{a}_{t-1}$ with embedding of the *previous action in the idiom definition*, thus emulating the tree fragment expansion. Outside the bounds of $\mathcal{I}$ (*i.e.* within the hole subtrees of $\mathcal{I}$) we use the actual $\boldsymbol{a}_{t-1}$ as usual.

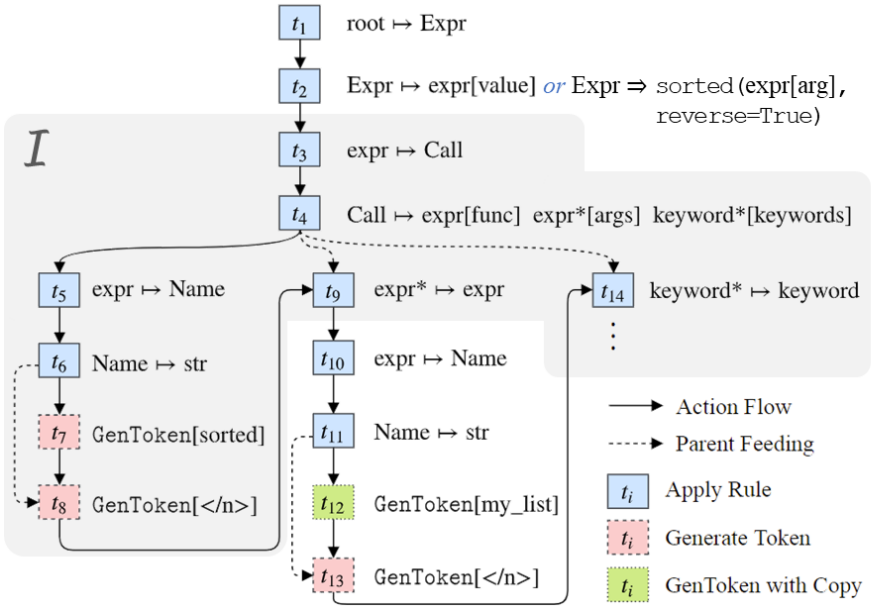


Figure 5.3: Decoding the AST `sorted(my_list,reverse=True)`, figure adapted from (Yin and Neubig 2017). Suppose an idiom $\mathcal{I}$ = `sorted(`$\boxed{\ell}$`,reverse=True)` is mined and added as an operator $\mathsf{op}_{\mathcal{I}}(\ell)$ to the grammar. At training time, PATOIS adjusts the cross-entropy objective at timestep t_2 to additionally allow $\mathsf{op}_{\mathcal{I}}$ as a valid production, with no change to further decoding. At inference time, if decoder emits an action $a_{t_2} = \textsc{ApplyRule}[\mathsf{op}_{\mathcal{I}}]$, PATOIS unrolls $\mathcal{I}$ on the fly by teacher-forcing the shaded portion of the AST generation.

5.5 Evaluation

Datasets We evaluate PATOIS on two semantic parsing datasets: Hearthstone (Ling et al. 2016) and Spider (Yu et al. 2018c).

Hearthstone is a dataset of 665 card descriptions from the trading card game of the same name, along with the implementations of their effects in Python using the game APIs. The descriptions act as NL specs X, and are on average 39.1 words long.

Spider is a dataset of 10,181 questions describing 5,693 unique SQL queries over 200 databases with multiple tables each. Each question pertains to a particular database, whose schema is given to the synthesizer. Database schemas do not overlap between the train and test splits, thus challenging the model to generalize across different domains. The questions are on average 13 words long and databases have on average 27.6 columns and 8.8 foreign keys.

Implementation We mine the idioms using the training split of each dataset. Thus PATOIS cannot indirectly overfit to the test set by learning its idioms, but it also cannot generalize beyond the idioms that occur in the training set. We run type-based MCMC (Section 5.3) for 10 iterations with $\alpha = 5$ and $d = 0.5$. After ranking (with either $\mathsf{Score}_{\mathsf{COV}}$ or $\mathsf{Score}_{\mathsf{CXE}}$) and filtering, we use K top-ranked

idioms to train the generative model. We ran ablation experiments with $K \in \{10, 20, 40, 80\}$.

As described in Section 5.4, for all our experiments we used a tree-based decoder with a pointer mechanism as the synthesizer f, which we implemented in PyTorch (Paszke et al. 2019). For the Hearthstone dataset, we use a bidirectional LSTM (Hochreiter and Schmidhuber 1997) to implement the description encoder $\hat{X} = f_{\text{enc}}(X)$, similarly to Yin and Neubig (2017). The word embeddings $\hat{\boldsymbol{x}}$ and hidden LSTM states $\boldsymbol{h}$ have dimension 256. The models are trained using the Adadelta optimizer (Zeiler 2012) with learning rate 1.0, $\rho = 0.95$, $\varepsilon = 10^{-6}$ for up to 2,600 steps with a batch size of 10.

For the Spider dataset, word embeddings $\hat{\boldsymbol{x}}$ have dimension 300, and hidden LSTM states $\boldsymbol{h}$ have dimension 256. The models are trained using the Adam optimizer (Kingma and Ba 2015) with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-9}$ for up to 40,000 steps with a batch size of 10. The learning rate warms up linearly up to 2.5×10^{-4} during the first 2,000 steps, and then decays polynomially by $(1 - t/T)^{-0.5}$ where T is the total number of steps. Each model configuration is trained on one NVIDIA GTX 1080 Ti GPU.

The Spider tasks additionally include the *database schema* as an input in the description. We follow a recent approach of embedding the schema using relation-aware self-attention within the encoder (Shin 2019). Specifically, we initialize a representation for each column, table, and word in the question, and then update these representations using 4 layers of relation-aware self-attention (Shaw et al. 2018) using a graph that describes the relations between columns and tables in the schema. See Chapter 4 for more details about the Spider schema encoder.

5.5.1 Experimental Results

In each configuration, we compare the performance of equivalent trained models on the same dataset with and without idiom-based training of PATOIS. For fairness, we show the performance of the same decoder implementation described in Section 5.4.1 as a baseline rather than the state-of-the-art results achieved by different approaches from the literature. Thus, our baseline is the decoder described in Section 5.4.1 trained with a regular cross-entropy objective rather than the PATOIS objective in Equation (5.5). Following prior work, we evaluate program generation as a semantic parsing task, and measure **(i)** exact match accuracy and BLEU scores for Hearthstone and **(ii)** exact match accuracy of program sketches for Spider.

Tables 5.2 and 5.3 show our ablation analysis of different configurations of PATOIS on the Hearthstone and Spider dev sets, respectively. Table 5.4 shows the test set results of the best model configuration for Hearthstone (the test instances for the Spider dataset are unreleased). As the results show, small numbers of idioms do not significantly change the exact match accuracy but improve BLEU score, and $K = 80$ gives a significant improvement in both the exact match accuracy and BLEU scores. The improvement is even more pronounced on the test set with 4.5% improvement in exact match accuracy and more than 4 BLEU points, which shows that mined training set idioms generalize well to the whole data distribution. As mentioned above, we compare only to the same baseline architecture for fairness, but PATOIS could also be easily implemented on top of the structural CNN decoder of Sun et al. (2019), the current state of the art on the Hearthstone dataset.

Table 5.2: Ablation tests on the Hearthstone dev set.

Model	K	Exact match	Sentence BLEU	Corpus BLEU
Baseline decoder	—	**0.197**	0.767	0.763
PATOIS, $\text{Score}_{\text{Cov}}$	10	0.151	0.781	**0.785**
	20	0.091	0.745	0.745
	40	0.167	0.765	0.764
	80	**0.197**	0.780	0.774
PATOIS, $\text{Score}_{\text{CXE}}$	10	0.151	0.780	0.783
	20	0.167	**0.787**	0.782
	40	0.182	0.773	0.770
	80	0.151	0.771	0.768

Table 5.3: Ablation tests on the Spider dev set.

Model	K	Exact match
Baseline decoder	—	0.395
PATOIS, $\text{Score}_{\text{Cov}}$	10	0.394
	20	0.379
	40	0.395
	80	0.407
PATOIS, $\text{Score}_{\text{CXE}}$	10	0.368
	20	0.382
	40	0.387
	80	**0.416**

Table 5.4: Test set results on Hearthstone (using the best configurations on the dev set).

Model	Exact match	Sentence BLEU	Corpus BLEU
Baseline	0.152	0.743	0.723
PATOIS	**0.197**	**0.780**	**0.766**

```
def __init__(self):
    super().__init__(ℓ0: str, ℓ1: int,
        CHARACTER_CLASS.ℓ3: id,
        CARD_RARITY.ℓ4: id, ℓ5?)
```

```
ℓ0: id = copy.copy(ℓ1: expr)
```

```
class ℓ0: id(ℓ1: id):
    def __init__(self):
        :
```

```
SELECT COUNT(ℓ0: col), ℓ1* WHERE ℓ2*
INTERSECT ℓ4?: sql EXCEPT ℓ5?: sql
```

```
WHERE ℓ0: col = $terminal
```

Figure 5.4: Five examples of commonly used idioms from the Hearthstone and Spider datasets.

Figure 5.4 shows some examples of idioms that were frequently used by the model. On Hearthstone, the most popular idioms involve common syntactic elements (*e.g.* class and function definitions) and domain-specific APIs commonly used in card implementations (*e.g.* `CARD_RARITY` enumerations or `copy.copy` calls). On Spider, they capture the most common combinations of SQL syntax, such as a `SELECT` query with a single `COUNT` column and optional `INTERSECT` or `EXCEPT` clauses. Notably, popular idioms are also often *big*: for instance, the first idiom in Figure 5.4 expands to a tree fragment with more than 20 nodes. Emitting it in a single step vastly simplifies the decoding process.

We further conducted qualitative experiments to analyze actual idiom usage by PATOIS on the

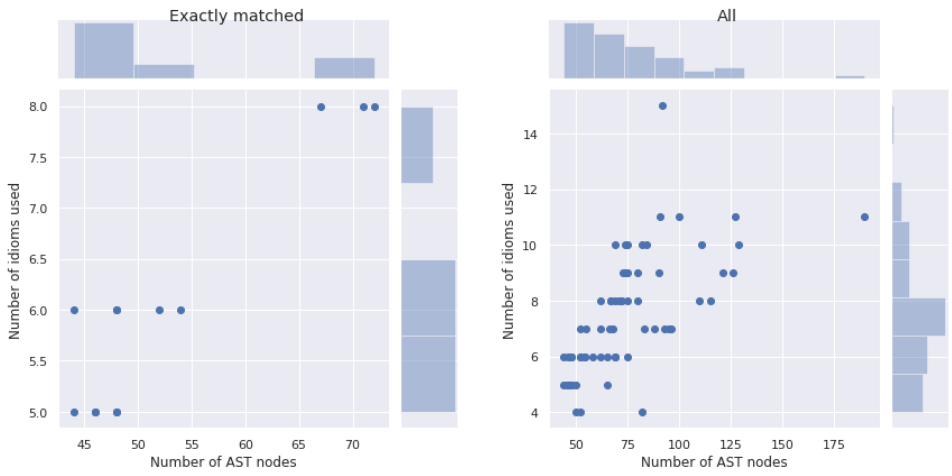


Figure 5.5: The distribution of used idioms in the inferred ASTs on the Hearthstone test set. *Left:* in the ASTs exactly matched with ground truth; *Right:* all ASTs.

Hearthstone test set. Figure 5.5 shows the distribution of idioms used in the inferred (not ground truth) ASTs. A typical program involves 7 idioms on average, or 6 for the programs that exactly match the ground truth. Despite the widespread usage of idioms, not all of the mined idioms $\widetilde{\mathcal{I}}$ were useful: only 51 out of $K = 80$ idioms appear in the inferred ASTs. This highlights the need for an end-to-end version of PATOIS where idiom mining would be directly optimized to benefit synthesis.

5.6 Related Work

Program synthesis & Semantic parsing Program synthesis from natural language has a long history in Programming Languages (PL) and Machine Learning (ML) communities. When an input specification is limited to natural language, the resulting problem can be considered *semantic parsing* (Liang 2016). There has been a lot of recent interest in applying recurrent sequence-based and tree-based neural networks to semantic parsing (Yin and Neubig 2017; Li et al. 2016; Dong and Lapata 2016; Jia and Liang 2016; Yin et al. 2018b). These approaches commonly use insights from the PL literature, such as grammar-based constraints to reduce the search space, non-deterministic training oracles to enable multiple executable interpretations of intent, and supervision from program execution. They typically either supervise the training on one or more golden programs, or use reinforcement learning to supervise the training from a neural program execution result (Neelakantan et al. 2017). Our PATOIS approach is applicable to any underlying neural semantic parsing model, as long as it is supervised by a corpus of golden programs. It is, however, most easily applicable to tree-based and graph-based models, which directly emit the AST of the target program. In this work we have evaluated PATOIS as applied on top of the sequence-to-tree decoder of Yin and Neubig (2017), and extended it with a novel training regime that teaches the decoder to emit idiom operators

in place of the idiomatic code fragments.

Sketch generation Two recent works (Dong and Lapata 2018; Murali et al. 2017) learn abstractions of the target program to compress and abstract the reasoning process of a neural synthesizer. Both of them split the generation process into *sketch generation* and *sketch completion*, wherein the first stage emits a partial tree/sequence (*i.e.* a *sketch* of the program) and the second stage fills in the holes in this sketch. While sketch generation is typically implemented with a neural model, sketch completion can be either a different neural model or a combinatorial search. In contrast to PATOIS, both works define the grammar of sketches manually by a deterministic *program abstraction* procedure and only allow a single top-level sketch for each program. In addition, an earlier work of Bonjak et al. (2017) also formulates program synthesis as sketch completion, but in their work program sketches are manually provided rather than learned. In PATOIS, we learn the abstractions (code idioms) automatically from a corpus and allow them to appear anywhere in the program, as is common in real-life programming.

Learning abstractions Recently, Ellis et al. (2018a) developed an Explore, Compress & Compile (EC^2) framework for automatically learning DSLs for program synthesis from I/O examples (such as the DSLs used by FlashFill (Gulwani 2011) and DeepCoder (Balog et al. 2016)). The workflow of EC^2 is similar to PATOIS, with three stages: **(a)** learn new DSL subroutines from a corpus of tasks, **(b)** train a recognition model that maps a task specification to a distribution over DSL operators as in DeepCoder (Balog et al. 2016), and **(c)** use these operators in a program synthesizer. PATOIS differs from EC^2 in three aspects: **(i)** we assume a natural language specification instead of examples, **(ii)** to handle NL specifications, our synthesizer is a neural semantic parser instead of enumerative search, and **(iii)** most importantly, we discover idioms that compress general-purpose languages instead of extending DSLs. Unlike for inductive synthesis DSLs such as FlashFill, the existence of *useful* DSL abstractions for general-purpose languages is not obvious, and our work is the first to demonstrate them.

Similar to this chapter, Iyer et al. (2019) developed a different approach of learning code idioms for semantic parsing. They mine the idioms using a variation of *byte-pair encoding* (BPE) compression extended to ASTs and greedily rewrite all the dataset ASTs in terms of the found idioms for training. While the BPE-based idiom mining is more computationally efficient than non-parametric Bayesian inference of PATOIS, introducing ASTs greedily tends to lose information about overlapping idioms, which we address in PATOIS using our novel training objective described in Section 5.4.2.

As described previously, our code idiom mining is an extension of the procedure developed by Allamanis et al. (Allamanis and Sutton 2014; Allamanis et al. 2018). They are the first to use the tree substitution grammar formalism and Bayesian inference to find non-trivial common idioms in a corpus of code. However, their problem formalization does not involve any application for the learned idioms beyond their explanatory power.

5.7 Discussion

Semantic parsing, or neural program synthesis from natural language, has made tremendous progress over the past years, but state-of-the-art models still struggle with program generation at multiple levels of abstraction. In this work, we present a framework that allows incorporating learned coding patterns from a corpus into the vocabulary of a neural synthesizer, thus enabling it to emit high-level or low-level program constructs interchangeably at each generation step. Our current instantiation, PATOIS, uses Bayesian inference to mine common code idioms, and employs a novel nondeterministic training regime to teach a tree-based generative model to optionally emit whole idiom fragments. Such dataset abstraction using idioms improves the performance of neural program synthesis.

PATOIS is only the first step toward learned abstractions in program synthesis. While code idioms often correlate with latent semantic concepts and our training regime allows the model to learn which idioms to use and in which context, our current method does not mine them with the intent to directly optimize their usefulness for generation. In future work, we want to alleviate this by jointly learning the mining and synthesis models, thus optimizing the idioms' usefulness for synthesis by construction. We also want to incorporate *program semantics* into the idiom definition, such as data flow patterns or natural language phrases from task specs.

Chapter 6

Conclusion

Program synthesis is a long-standing problem in computer science with significant practical applications. In recent years, program synthesis with deep learning, or neural program synthesis, has arisen as a promising direction and catalyzed progress in the field. This dissertation has considered three challenges in machine learning that also apply to neural program synthesis, and addressed them in several concrete instantiations of neural program synthesis.

In the first part, we looked at program synthesis from input-output examples as the specification. First, we presented a technique to improve synthesis of imperative Karel programs by breaking down the problem into two simpler parts of predicting execution traces from input-output examples and then predicting the program from the traces (Chapter 2; addressing issues of *embodiment* with imperative programs through the *specification encoder*). Next, we showed how synthetic data generation for program synthesis can be trickier than assumed by previous work, as the exact procedure used for random sampling of the training data has significant effect on the ability of the resulting model to generalize (Chapter 3; addressing *generalization* through the *training procedure*). In the second part, we examine program synthesis from natural language specifications, or semantic parsing. We show that relation-aware self-attention, by facilitating learned interactions within the input while also incorporating structured information, enables better generalization when synthesizing database queries from a multi-modal specification containing a natural language question and a database schema (Chapter 4; addressing *generalization* through the *specification encoder*). We also demonstrate a method to help a neural program synthesizer switch between high-level and low-level reasoning by incorporating learned *idioms* (Chapter 5; addressing *abstraction* through the *program decoder*).

Reaching the goal of creating computers that program themselves will require many advances beyond the contributions in this dissertation. Some of these advances will undoubtedly originate from unanticipated places with little direct relation to program synthesis; nevertheless, our work also suggests many open questions and future directions to bring us closer to the goal. Here are some of them.

Human factors. The overall paradigm for program synthesis envisioned in this dissertation is the following: the system receives a program specification, and then it does its best to generate a

program that fulfills it. If all goes well, the system would correctly understand the specification, and generate the right program. Unfortunately, the capabilities of current neural program synthesis is not sufficient so that all would go well most of the time, but there are no explicit provisions for recovering from errors.

First, it may be difficult for the user to understand the generated program sufficiently enough to realize that there has been an error. The easiest setting is when the user is expected to be a proficient programmer already, and the system generates first drafts for the user to review and correct. For program synthesis from input-output examples, a user can still double-check whether the program generates the expected output when applied to additional inputs, without knowing the programming language. However, this can be challenging when there are many inputs (like rows in a spreadsheet) but the program may fail only on a small number. More challenging is the setting of generating a database query from natural language. Without knowledge of SQL, the user cannot check the generated query for errors. Since the user presumably does not know the expected answer for the question, simply displaying the results of the query would not help much either.

Therefore, it would be important to empower the user to understand what the generated program does and provide a correction if needed. Using rules to generate a natural language explanation are likely sufficient for simple SQL queries, but maintaining conciseness and fluency becomes more challenging for longer and complex queries. If the user provides an explanation for the error, we need additional mechanisms to make the fix based on the explanation. Prior work has explained some of these questions (for example, Elgohary et al. (2020)) and we will need to integrate everything together to obtain a useful system.

Learning and using abstractions. In Chapter 5, we showed that a program decoder can successfully make use of common idioms to generate programs. We first create a set of candidate idioms, and then encourage the decoder to use them; this is the same in the concurrent work Iyer et al. (2019). When training the decoder, we cannot change the set of idioms based on some form of feedback from the decoder. If this were possible, we may be able to choose more useful idioms that the decoder can use more profitably.

More generally, there are many kinds of abstractions used in programming languages, ranging from functions and classes to more implicit constructs like design patterns. Existing work is largely limited to learning how to use existing functions, but cannot create new ones when needed. The ability to create and use these kinds of abstractions will pave the way for synthesizing larger and more complex programs.

Improved structured decoders. In Chapters 4 and 5, we used Yin and Neubig (2017) as the basis for the program decoder. Structured decoders often work better compared to generating programs as a sequence of tokens, by freeing the model from needing to enforce syntactic correctness of its outputs, and containing inductive biases suited to program generation.

However, they do not enforce other properties needed for correctness of the output other than syntactic correctness. They operate in a fixed and inflexible order from the beginning to the end. They cannot review or edit their past outputs. In contrast, human programmers will write different

parts of a program in a non-linear order, and make edits to past decisions when errors become apparent.

Better synthetic data generation. We showed the importance of synthetic data generation in Chapter 3, wherein changes to *salient random variables* had an outsized impact on the model's ability to generalize to new inputs. We found that straightforward ways to generate data can lead to unusually skewed distributions in these salient random variables, like program length and nesting depth, but our methods to re-balance the distributions are based on rejection sampling which can be inefficient. We also need to manually define the salient random variables. Adversarially training a data generation model to generate unusual examples for the program synthesis model might be one approach to enable greater efficiency and avoid the need to specify variables explicitly.

Bibliography

Aggarwal, Charu C. and Jiawei Han, eds. (2014). *Frequent Pattern Mining*. en. Springer International Publishing. ISBN: 978-3-319-07820-5. DOI: 10.1007/978-3-319-07821-2.

Allamanis, Miltiadis, Earl T. Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton (July 2018). "Mining Semantic Loop Idioms". In: *IEEE Transactions on Software Engineering* 44.7, pp. 651–668. ISSN: 2326-3881. DOI: 10.1109/TSE.2018.2832048.

Allamanis, Miltiadis and Charles Sutton (Nov. 2014). "Mining Idioms from Source Code". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, pp. 472–483. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635901.

Andreas, Jacob, Marcus Rohrbach, Trevor Darrell, and Dan Klein (June 2016). "Learning to Compose Neural Networks for Question Answering". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, pp. 1545–1554. DOI: 10.18653/v1/N16-1181.

Androutsopoulos, I., G. D. Ritchie, and P. Thanisch (Mar. 1995). "Natural Language Interfaces to Databases - An Introduction". In: *arXiv:cmp-lg/9503016*. URL: http://arxiv.org/abs/cmp-lg/9503016.

Antol, Stanislaw, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh (Dec. 2015). "VQA: Visual Question Answering". In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2425–2433. DOI: 10.1109/ICCV.2015.279.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1409.0473.

Balog, Matej, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow (Nov. 2016). "DeepCoder: Learning to Write Programs". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=ByldLrqlx.

Bhupatiraju, Surya, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli (Apr. 2017). "Deep API Programmer: Learning to Program with APIs". In: *arXiv:1704.04327 [cs]*. URL: http://arxiv.org/abs/1704.04327.

Bonjak, Matko, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel (July 2017). "Programming with a Differentiable Forth Interpreter". en. In: *International Conference on Machine Learning*. Chap. Machine Learning, pp. 547–556. URL: http://proceedings.mlr.press/v70/bosnjak17a.html.

Bousmalis, Konstantinos, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke (May 2018). "Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4243–4250. DOI: 10.1109/ICRA.2018.8460875.

Brockschmidt, Marc, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov (Sept. 2018). "Generative Code Modeling with Graphs". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=Bke4KsA5FX.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (June 2020). "Language Models Are Few-Shot Learners". In: *arXiv:2005.14165 [cs]*. URL: http://arxiv.org/abs/2005.14165.

Bunel, Rudy, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli (Feb. 2018). "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=H1Xw62kRZ.

Buolamwini, Joy and Timnit Gebru (Jan. 2018). "Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification". en. In: *Conference on Fairness, Accountability and Transparency*. Chap. Machine Learning, pp. 77–91. URL: http://proceedings.mlr.press/v81/buolamwini18a.html.

Cai, Jonathon, Richard Shin, and Dawn Song (Nov. 2016). "Making Neural Programming Architectures Generalize via Recursion". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=BkbY4psgg.

Chen, Xinyun, Chang Liu, and Dawn Song (Sept. 2018). "Execution-Guided Neural Program Synthesis". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=H1gfOiAqYm.

Christiano, Paul, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba (Oct. 2016). "Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model". In: *arXiv:1610.03518 [cs]*. URL: http://arxiv.org/abs/1610.03518.

Cohn, Trevor, Phil Blunsom, and Sharon Goldwater (2010). "Inducing Tree-Substitution Grammars". In: *Journal of Machine Learning Research* 11.Nov, pp. 3053–3096. ISSN: ISSN 1533-7928. URL: http://www.jmlr.org/papers/v11/cohn10b.html.

Devlin, Jacob, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli (2017a). "Neural Program Meta-Induction". In: *Advances in Neural Information Processing Systems*

30. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 2080–2088. URL: http://papers.nips.cc/paper/6803-neural-program-meta-induction.pdf.

Devlin, Jacob, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli (July 2017b). "RobustFill: Neural Program Learning under Noisy I/O". en. In: *International Conference on Machine Learning*, pp. 990–998. URL: http://proceedings.mlr.press/v70/devlin17a.html.

Dong, Li and Mirella Lapata (Aug. 2016). "Language to Logical Form with Neural Attention". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 33–43. DOI: 10.18653/v1/P16-1004.

— (May 2018). "Coarse-to-Fine Decoding for Neural Semantic Parsing". In: *arXiv:1805.04793 [cs]*. URL: http://arxiv.org/abs/1805.04793.

Du, Tao, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik (Dec. 2018). "InverseCSG: Automatic Conversion of 3D Models to CSG Trees". In: *ACM Transactions on Graphics* 37.6, 213:1–213:16. ISSN: 0730-0301. DOI: 10.1145/3272127.3275006.

Elgohary, Ahmed, saghar Hosseini, and Ahmed Hassan Awadallah (July 2020). "Speak to Your Parser: Interactive Text-to-SQL with Natural Language Feedback". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, pp. 2065–2077. URL: https://www.aclweb.org/anthology/2020.acl-main.187.

Ellis, Kevin, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum (2018a). "Learning Libraries of Subroutines for Neurally Guided Bayesian Program Induction". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., pp. 7805–7815. URL: http://papers.nips.cc/paper/8006-learning-libraries-of-subroutines-for-neurallyguided-bayesian-program-induction.pdf.

Ellis, Kevin, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama (2019). "Write, Execute, Assess: Program Synthesis with a REPL". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 9169–9178. URL: http://papers.nips.cc/paper/9116-write-execute-assess-program-synthesis-with-a-repl.pdf.

Ellis, Kevin, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum (2018b). "Learning to Infer Graphics Programs from Hand-Drawn Images". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., pp. 6059–6068. URL: http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf.

Finegan-Dollak, Catherine, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev (2018). "Improving Text-to-SQL Evaluation Method-

ology". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, pp. 351–360. URL: http://aclweb.org/anthology/P18-1033.

Fukushima, Kunihiko (Apr. 1980). "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". en. In: *Biological Cybernetics* 36.4, pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251.

Gal, Yarin and Zoubin Ghahramani (2016). "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., pp. 1019–1027. URL: http://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.pdf.

Ganin, Yaroslav, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals (July 2018). "Synthesizing Programs for Images Using Reinforced Adversarial Learning". en. In: *International Conference on Machine Learning*. Chap. Machine Learning, pp. 1666–1675. URL: http://proceedings.mlr.press/v80/ganin18a.html.

Gaunt, Alexander L., Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow (Aug. 2016). "TerpreT: A Probabilistic Programming Language for Program Induction". In: *arXiv:1608.04428 [cs]*. URL: http://arxiv.org/abs/1608.04428.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

Graves, Alex, Greg Wayne, and Ivo Danihelka (Dec. 2014). "Neural Turing Machines". In: *arXiv:1410.5401 [cs]*. URL: http://arxiv.org/abs/1410.5401.

Graves, Alex, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwiska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis (Oct. 2016). "Hybrid Computing Using a Neural Network with Dynamic External Memory". en. In: *Nature* 538.7626, pp. 471–476. ISSN: 0028-0836. DOI: 10.1038/nature20101.

Green, Cordell (May 1969). "Application of Theorem Proving to Problem Solving". In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., pp. 219–239.

Gregor, Karol, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra (June 2015). "DRAW: A Recurrent Neural Network For Image Generation". en. In: *International Conference on Machine Learning*. Chap. Machine Learning, pp. 1462–1471. URL: http://proceedings.mlr.press/v37/gregor15.html.

Gulwani, Sumit (Jan. 2011). "Automating String Processing in Spreadsheets Using Input-Output Examples". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423.

Gulwani, Sumit, Oleksandr Polozov, and Rishabh Singh (July 2017). "Program Synthesis". English. In: *Foundations and Trendső in Programming Languages* 4.1-2, pp. 1–119. ISSN: 2325-1107, 2325-1131. DOI: 10.1561/2500000010.

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

Hu, Ronghang, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko (Oct. 2017). "Learning to Reason: End-to-End Module Networks for Visual Question Answering". In: *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 804–813. DOI: 10.1109/ICCV.2017.93.

Iyer, Srinivasan, Alvin Cheung, and Luke Zettlemoyer (Nov. 2019). "Learning Programmatic Idioms for Scalable Semantic Parsing". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, pp. 5426–5435. DOI: 10.18653/v1/D19-1545.

Iyer, Srinivasan, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer (July 2017). "Learning a Neural Semantic Parser from User Feedback". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 963–973. DOI: 10.18653/v1/P17-1089.

Jia, Robin and Percy Liang (Aug. 2016). "Data Recombination for Neural Semantic Parsing". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 12–22. DOI: 10.18653/v1/P16-1002.

Joulin, Armand and Tomas Mikolov (2015). "Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., pp. 190–198. URL: http://papers.nips.cc/paper/5857-inferring-algorithmic-patterns-with-stack-augmented-recurrent-nets.pdf.

Kaiser, ukasz and Ilya Sutskever (Nov. 2015). "Neural GPUs Learn Algorithms". In: *arXiv:1511.08228 [cs]*. URL: http://arxiv.org/abs/1511.08228.

Kalyan, Ashwin, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani (Feb. 2018). "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=rywDjg-RW.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1412.6980.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (May 2017). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6, pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

Kurach, Karol, Marcin Andrychowicz, and Ilya Sutskever (Feb. 2016). "Neural Random-Access Machines". In: *arXiv:1511.06392 [cs]*. URL: http://arxiv.org/abs/1511.06392.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (Dec. 1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). "Deep Learning". en. In: *Nature* 521.7553, pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539.

Leveson, N.G. and C.S. Turner (July 1993). "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7, pp. 18–41. ISSN: 1558-0814. DOI: 10.1109/MC.1993.274940.

Li, Fei and H. V. Jagadish (Sept. 2014). "Constructing an Interactive Natural Language Interface for Relational Databases". In: *Proceedings of the VLDB Endowment* 8.1, pp. 73–84. ISSN: 2150-8097. DOI: 10.14778/2735461.2735468.

Li, Yujia, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel (May 2016). "Gated Graph Sequence Neural Networks". In: *International Conference on Learning Representations*. URL: http://arxiv.org/abs/1511.05493.

Liang, Percy (Aug. 2016). "Learning Executable Semantic Parsers for Natural Language Understanding". In: *Communications of the ACM* 59.9, pp. 68–76. ISSN: 0001-0782. DOI: 10.1145/2866568.

Liang, Percy, Michael I. Jordan, and Dan Klein (June 2010). "Type-Based MCMC". In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Los Angeles, California: Association for Computational Linguistics, pp. 573–581. URL: https://www.aclweb.org/anthology/N10-1082.

Ling, Wang, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomá Koiský, Fumin Wang, and Andrew Senior (Aug. 2016). "Latent Predictor Networks for Code Generation". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 599–609. DOI: 10.18653/v1/P16-1057.

Lopes, Raphael Gontijo, David Ha, Douglas Eck, and Jonathon Shlens (Oct. 2019). "A Learned Representation for Scalable Vector Graphics". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 7929–7938. DOI: 10.1109/ICCV.2019.00802.

Manna, Zohar and Richard J. Waldinger (Mar. 1971). "Toward Automatic Program Synthesis". In: *Communications of the ACM* 14.3, pp. 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568.

McCulloch, Warren S. and Walter Pitts (Dec. 1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity". en. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259.

Murali, Vijayaraghavan, Letao Qi, Swarat Chaudhuri, and Chris Jermaine (Mar. 2017). "Neural Sketch Learning for Conditional Program Generation". In: *arXiv:1703.05698 [cs]*. URL: http://arxiv.org/abs/1703.05698.

Neelakantan, Arvind, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei (2017). "Learning a Natural Language Interface with Neural Programmer". In: *5th International Confer-*

ence on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net. URL: https://openreview.net/forum?id=ry2YOrcge.

Oberg, James (Dec. 1999). *Why the Mars Probe Went off Course - IEEE Spectrum*. en. URL: https://spectrum.ieee.org/aerospace/robotic-exploration/why-the-mars-probe-went-off-course.

Parisotto, Emilio, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli (Nov. 2016). "Neuro-Symbolic Program Synthesis". In: *arXiv:1611.01855 [cs]*. URL: http://arxiv.org/abs/1611.01855.

Parker, David (1985). *Learning Logic*. Technical Report 47. Cambridge, MA: Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8026–8037. URL: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pattis, Richard E. (1981). *Karel the Robot: A Gentle Introduction to the Art of Programming*. 1st. USA: John Wiley & Sons, Inc. ISBN: 978-0-471-08928-5.

Peng, Xue Bin, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel (May 2018). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3803–3810. DOI: 10.1109/ICRA.2018.8460528.

Pinto, Lerrel, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel (June 2018). "Asymmetric Actor Critic for Image-Based Robot Learning". In: *Robotics: Science and Systems XIV*. Vol. 14. ISBN: 978-0-9923747-4-7. URL: http://www.roboticsproceedings.org/rss14/p08.html.

Polozov, Oleksandr and Sumit Gulwani (Oct. 2015). "FlashMeta: A Framework for Inductive Program Synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 107–126. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814310.

Popescu, Ana-Maria, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates (2004). "Modern Natural Language Interfaces to Databases: Composing Statistical Parsing with Semantic Tractability". In: *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. URL: http://aclweb.org/anthology/C04-1021.

Popper, Nathaniel (Aug. 2012). *Knight Capital Says Trading Glitch Cost It $440 Million*. en. URL: https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/.

Post, Matt and Daniel Gildea (Aug. 2009). "Bayesian Learning of a Tree Substitution Grammar". In: *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*. Suntec, Singapore: Association for Computational Linguistics, pp. 45–48. URL: https://www.aclweb.org/anthology/P09-2012.

Rabinovich, Maxim, Mitchell Stern, and Dan Klein (July 2017). "Abstract Syntax Networks for Code Generation and Semantic Parsing". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 1139–1149. DOI: 10.18653/v1/P17-1105.

Reed, Scott and Nando de Freitas (Feb. 2016). "Neural Programmer-Interpreters". In: *arXiv:1511.06279 [cs]*. URL: http://arxiv.org/abs/1511.06279.

Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6, pp. 386–408. ISSN: 1939-1471(Electronic),0033-295X(Print). DOI: 10.1037/h0042519.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). "Learning Representations by Back-Propagating Errors". en. In: *Nature* 323.6088, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei (Dec. 2015). "ImageNet Large Scale Visual Recognition Challenge". en. In: *International Journal of Computer Vision* 115.3, pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y.

Sabour, Sara, Nicholas Frosst, and Geoffrey E Hinton (2017). "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 3856–3866. URL: http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf.

See, Abigail, Peter J. Liu, and Christopher D. Manning (July 2017). "Get To The Point: Summarization with Pointer-Generator Networks". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 1073–1083. DOI: 10.18653/v1/P17-1099.

Sethuraman, Jayaram (1994). "A Constructive Definition of Dirichlet Priors". In: *Statistica Sinica* 4.2, pp. 639–650. ISSN: 1017-0405. URL: https://www.jstor.org/stable/24305538.

Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani (2018). "Self-Attention with Relative Position Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, pp. 464–468. DOI: 10.18653/v1/N18-2074.

Shi, Tianze, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen (Sept. 2018). "IncSQL: Training Incremental Text-to-SQL Parsers with Non-Deterministic Oracles". In: *arXiv:1809.05054 [cs]*. URL: http://arxiv.org/abs/1809.05054.

Shin, Richard (June 2019). "Encoding Database Schemas with Relation-Aware Self-Attention for Text-to-SQL Parsers". In: *arXiv:1906.11790 [cs, stat]*. URL: http://arxiv.org/abs/1906.11790.

Shin, Richard, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov (June 2019). "Program Synthesis and Semantic Parsing with Learned Code Idioms". In: *arXiv:1906.10816 [cs, stat]*. URL: http://arxiv.org/abs/1906.10816.

Shin, Richard, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song (Sept. 2018a). "Synthetic Datasets for Neural Program Synthesis". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=ryeOSnAqYm.

Shin, Richard, Illia Polosukhin, and Dawn Song (2018b). "Improving Neural Program Synthesis with Inferred Execution Traces". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., pp. 8917–8926. URL: http://papers.nips.cc/paper/8107-improving-neural-program-synthesis-with-inferred-execution-traces.pdf.

Shrivastava, Ashish, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb (July 2017). "Learning from Simulated and Unsupervised Images through Adversarial Training". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2242–2251. DOI: 10.1109/CVPR.2017.241.

Solar-Lezama, Armando (Oct. 2013). "Program Sketching". en. In: *International Journal on Software Tools for Technology Transfer* 15.5, pp. 475–495. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0249-7.

Spider: Yale Semantic Parsing and Text-to-SQL Challenge (2019). Accessed 2019-03-02. URL: https://yale-lily.github.io/spider.

Sukhbaatar, Sainbayar, arthur szlam, Jason Weston, and Rob Fergus (2015). "End-To-End Memory Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., pp. 2440–2448. URL: http://papers.nips.cc/paper/5846-end-to-end-memory-networks.pdf.

Sun, Zeyu, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang (July 2019). "A Grammar-Based Structural CNN Decoder for Code Generation". en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01, pp. 7055–7062. ISSN: 2374-3468. DOI: 10.1609/aaai.v33i01.33017055.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., pp. 3104–3112. URL: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

Sutton, Richard S., Doina Precup, and Satinder Singh (Aug. 1999). "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning". en. In: *Artificial*

Intelligence 112.1, pp. 181–211. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(99)00052-1.

Tang, Lappoon R. and Raymond J. Mooney (Oct. 2000). "Automated Construction of Database Interfaces: Intergrating Statistical and Relational Learning for Semantic Parsing". In: *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. Hong Kong, China: Association for Computational Linguistics, pp. 133–141. DOI: 10.3115/1117794.1117811.

Teh, Yee Whye and Michael I. Jordan (Apr. 2010). *Hierarchical Bayesian Nonparametric Models with Applications*. en. DOI: 10.1017/CBO9780511802478.006.

Tian, Yonglong, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu (Sept. 2018). "Learning to Infer and Execute 3D Shape Programs". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=rylNH20qFQ.

Vaswani, Ashish, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, ukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit (Mar. 2018). "Tensor2Tensor for Neural Machine Translation". In: *arXiv:1803.07416 [cs, stat]*. URL: http://arxiv.org/abs/1803.07416.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 5998–6008. URL: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Waldinger, Richard J. and Richard C. T. Lee (May 1969). "PROW: A Step toward Automatic Program Writing". In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., pp. 241–252.

Wang, Bailin, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson (July 2020). "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, pp. 7567–7578. URL: https://www.aclweb.org/anthology/2020.acl-main.677.

Wang, Ke, Rishabh Singh, and Zhendong Su (Feb. 2018). "Dynamic Neural Program Embeddings for Program Repair". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=BJuWrGW0Z.

Werbos, Paul (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University.

Weston, Jason, Sumit Chopra, and Antoine Bordes (Nov. 2015). "Memory Networks". In: *arXiv:1410.3916 [cs, stat]*. URL: http://arxiv.org/abs/1410.3916.

A Large Annotated Semantic Parsing Corpus for Developing Natural Language Interfaces. (2019). *A Large Annotated Semantic Parsing Corpus for Developing Natural Language Interfaces.: Salesforce/WikiSQL*. Accessed 2019-03-02. URL: https://github.com/salesforce/WikiSQL.

World Bank (2018a). *Individuals Using the Internet (% of Population)*. URL: https://data.worldbank.org/indicator/IT.NET.USER.ZS.

— (2018b). *Mobile Cellular Subscriptions (per 100 People)*. URL: https://data.worldbank.org/indicator/IT.CEL.SETS.P2.

Xu, Xiaojun, Chang Liu, and Dawn Song (Nov. 2017). "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning". In: *arXiv:1711.04436 [cs]*. URL: http://arxiv.org/abs/1711.04436.

Yaghmazadeh, Navid, Yuepeng Wang, Isil Dillig, and Thomas Dillig (Oct. 2017). "SQLizer: Query Synthesis from Natural Language". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA, 63:1–63:26. DOI: 10.1145/3133887.

Yin, Pengcheng, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig (May 2018a). "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 476–486. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196408.

Yin, Pengcheng and Graham Neubig (2017). "A Syntactic Neural Model for General-Purpose Code Generation". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 440–450. DOI: 10.18653/v1/P17-1041.

Yin, Pengcheng, Chunting Zhou, Junxian He, and Graham Neubig (July 2018b). "StructVAE: Tree-Structured Latent Variable Models for Semi-Supervised Semantic Parsing". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, pp. 754–765. DOI: 10.18653/v1/P18-1070.

Yu, Tao, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev (2018a). "TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL Generation". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, pp. 588–594. DOI: 10.18653/v1/N18-2093.

Yu, Tao, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev (2018b). "SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, pp. 1653–1663. URL: http://aclweb.org/anthology/D18-1193.

Yu, Tao, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev (2018c). "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, pp. 3911–3921. URL: http://aclweb.org/anthology/D18-1425.

Zaremba, Wojciech and Ilya Sutskever (Feb. 2015). "Learning to Execute". In: *arXiv:1410.4615 [cs]*. URL: http://arxiv.org/abs/1410.4615.

Zeiler, Matthew D. (Dec. 2012). "ADADELTA: An Adaptive Learning Rate Method". In: *arXiv:1212.5701 [cs]*. URL: http://arxiv.org/abs/1212.5701.

Zelle, John M. and Raymond J. Mooney (Aug. 1996). "Learning to Parse Database Queries Using Inductive Logic Programming". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*. AAAI'96. Portland, Oregon: AAAI Press, pp. 1050–1055. ISBN: 978-0-262-51091-2.

Zettlemoyer, Luke S. and Michael Collins (July 2005). "Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars". In: *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. UAI'05. Edinburgh, Scotland: AUAI Press, pp. 658–666. ISBN: 978-0-9749039-1-0.

Zhong, Victor, Caiming Xiong, and Richard Socher (Aug. 2017). "Seq2SQL: Generating Structured Queries from Natural Language Using Reinforcement Learning". In: *arXiv:1709.00103 [cs]*. URL: http://arxiv.org/abs/1709.00103.

Zohar, Amit and Lior Wolf (Jan. 2019). "Automatic Program Synthesis of Long Programs with a Learned Garbage Collector". In: *arXiv:1809.04682 [cs, stat]*. URL: http://arxiv.org/abs/1809.04682.

Appendix A

Synthetic Datasets

A.1 Salient Variable Homogenization Algorithm

The following is a more formal description of the Algorithm described in Section 3.2, as well as proofs of correctness, and an investigation of the ϵ parameter's practical effect.

A.1.1 Full Pseudocode

Let $\mathbb{S}$ be some space that is sampled by some original distribution $q : \mathbb{S} \to [0, 1]$. Let $\mathbb{X}$ be the space of a salient variable which is calculated by $\nu : \mathbb{S} \to \mathbb{X}$; it is a finite set. Let ε be our tolerance.

procedure SAMPLEDHOMOGENIZE($q : \mathbb{S} \to [0, 1]$, $\nu : \mathbb{S} \to \mathbb{X}$, $\varepsilon \in \mathbb{R}$, $n \in \mathbb{N}$)
  $C \leftarrow \{x \to 0 : x \in \mathbb{X}\}$ ▷ Set up a dictionary of counts of each value of $\mathbb{X}$
  $t \leftarrow 0$ ▷ The total number of samples seen
  $\mathcal{D} \leftarrow \{\}$ ▷ $\mathcal{D}$ is originally an empty multiset
  while $|\mathcal{D}| < n$ **do**
    $s \leftarrow$ a sample from q
    $C[\nu(s)] \leftarrow C[\nu(s)] + 1$
    $t \leftarrow t + 1$
    $p_{\min} \leftarrow \frac{\min_{x \in \mathbb{X}} C[x]}{t}$
    $p_{curr} \leftarrow \frac{C[\nu(s)]}{t}$
    $g \leftarrow \frac{p_{\min} + \varepsilon}{p_{curr} + \varepsilon}$
    $h \leftarrow \begin{cases} 1 & \text{with probability } g \\ 0 & \text{with probability } 1 - g \end{cases}$
    if $h = 1$ **then**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{s\}$
    end if
  end while
  return $\mathcal{D}$
end procedure

A.1.2 Proof of Correctness

Let the initial sampling distribution be q and the resulting distribution be r. We use the notation $P_r[X = x]$ to refer to the probability that $X = x$ given that S is sampled from the distribution r. $C[x]$ refers to the count for the salient variable value $x \in \mathbb{X}$ among past samples from q, as defined in SAMPLEDHOMOGENIZE.

Theorem. *The Salient Variable Homogenization algorithm produces samples from a distribution close to uniform. Formally, after* $\frac{48 \log \frac{2|\mathbb{X}|}{\delta}}{p_m |\mathbb{X}|^2 \xi^2}$ *samples are drawn from distribution* q*, we have that the resulting homogenized distribution satisfies*

$$\left| P_r[X = x] - \frac{1}{|\mathbb{X}|} \right| \leq \xi$$

with probability at least $1 - \delta$*, where* $p_m = \min_{x \in \mathbb{X}} P_q[X = x] > 0$ *and* ε *has been set to* 0*.*

Proof. We can simplify the probability as

$$P_r[X = x] = \frac{P_q[X = x]/C[x]}{\sum_{z \in \mathbb{X}} P_q[X = z]/C[z]}$$

(for $\varepsilon = 0$; see Probability Simplification Lemma below).

Let $\alpha = \frac{\xi |\mathbb{X}|}{4}$. We have that $n > \frac{3 \log \frac{2|\mathbb{X}|}{\delta}}{\alpha^2 p_m}$ by assumption and substituting in α. By the Count Bounding Lemma we have that $\left| \frac{nP_q[X=x]}{C[x]} - 1 \right| \leq \alpha$ for all x with probability at least $1 - \delta$. The following computations assume $\left| \frac{nP_q[X=x]}{C[x]} - 1 \right| \leq \alpha$ for all x and thus are valid with probability $1 - \delta$.

We have $\frac{nP_q[X=x]}{C[x]} \in [1 - \alpha, 1 + \alpha]$. We also have

$$\begin{aligned}
\left| \frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} nP[X = z]/C[z] - 1 \right| &= \left| \frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} (nP[X = z]/C[z] - 1) \right| \\
&=\leq \frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} |nP[X = z]/C[z] - 1| \\
&=\leq \frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} \alpha \\
&= \alpha
\end{aligned}$$

and thus $\frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} \frac{nP[X=z]}{C[z]} \in [1 - \alpha, 1 + \alpha]$.

Combining the previous two ranges via a division, we have

$$|\mathbb{X}| P_r[X = x] = \frac{\frac{nP_q[X=x]}{C[x]}}{\frac{1}{|\mathbb{X}|} \sum_{z \in \mathbb{X}} \frac{nP[X=z]}{C[z]}} \in \left[\frac{1 - \alpha}{1 + \alpha}, \frac{1 + \alpha}{1 - \alpha} \right]$$

We have that $\frac{1-\alpha}{1+\alpha} = 1 - \frac{2\alpha}{1+\alpha} \geq 1 - 2\alpha \geq 1 - 4\alpha$ and $\frac{1+\alpha}{1-\alpha} = 1 + \frac{2\alpha}{1-\alpha} = 1 + \frac{4\alpha}{2-2\alpha} \leq 1 + 4\alpha$ since $2 - 2\alpha > 1$ for small α. Thus, we have that $\left[\frac{1-\alpha}{1+\alpha}, \frac{1+\alpha}{1-\alpha}\right] \subseteq [1 - 4\alpha, 1 + 4\alpha]$ and therefore we have

$$|\mathbb{X}|P_r[X = x] \in [1 - 4\alpha, 1 + 4\alpha]$$

we thus have that $\left|P_r[X = x] - \frac{1}{|\mathbb{X}|}\right| \leq \frac{4\alpha}{|\mathbb{X}|} = \xi$, completing our proof. □

Lemma (Count Bounding)**.** *We have that if* $n > \frac{3 \log \frac{2|\mathbb{X}|}{\delta}}{\alpha^2 p_m}$ *samples have been drawn from* q *that*

$$P\left[\exists x \in \mathbb{X}, \left|\frac{nP_q[X = x]}{C[x]} - 1\right| \geq \alpha\right] \leq \delta$$

where $p_m = \min_{x \in \mathbb{X}} P_q[X = x]$

Proof. We can model $C[x]$ as a sum of n independent Bernoulli trials with probability $P_q[X = x]$. Using Chernoff bound, we have that

$$P[C[x] \geq nP_q[X = x](1 + \alpha)] \leq e^{-\alpha^2 nP_q[X=x]/3}$$

and

$$P[C[x] \geq nP_q[X = x](1 - \sqrt{2/3}\alpha)] \leq e^{-\alpha^2 nP_q[X=x]/3}$$

Thus, we have by union bound that

$$P\left[\frac{nP_q[X = x]}{C[x]} \geq \frac{1}{1 - \sqrt{2/3}\alpha} \vee \frac{nP_q[X = x]}{C[x]} \leq \frac{1}{1 + \alpha}\right] \leq 2e^{-\alpha^2 nP_q[X=x]/3}$$

For $\alpha \leq \sqrt{3/2} - 1$ we have that $\frac{1}{1-\sqrt{2/3}\alpha} = 1 + \frac{\sqrt{2/3}\alpha}{1-\sqrt{2/3}\alpha} = 1 + \frac{\alpha}{\sqrt{3/2}-\alpha} \leq 1 + \alpha$ and $\frac{1}{1+\alpha} = 1 - \frac{\alpha}{1+\alpha} \geq 1 - \alpha$. Thus, we can restate the previous inequality as

$$P\left[\frac{nP_q[X = x]}{C[x]} \geq 1 + \alpha \vee \frac{nP_q[X = x]}{C[x]} \leq 1 - \alpha\right] \leq 2e^{-\alpha^2 nP_q[X=x]/3}$$

or in other words

$$P\left[\left|\frac{nP_q[X = x]}{C[x]} - 1\right| \geq \alpha\right] \leq 2e^{-\alpha^2 nP_q[X=x]/3}$$

Thus we can bound the RHS as

$$2e^{-\alpha^2 nP_q[X=x]/3} \leq 2e^{-\alpha^2 np_m/3} < 2e^{-\alpha^2 \frac{3 \log \frac{2|\mathbb{X}|}{\delta}}{\alpha^2 p_m} p_m/3} = 2e^{-\log \frac{2|\mathbb{X}|}{\delta}} = \frac{\delta}{|\mathbb{X}|}$$

where the first inequality is since $P_q[X = x] \geq p_m$ and the second is since $n > \frac{3 \log \frac{2|\mathbb{X}|}{\delta}}{\alpha^2 p_m}$. We can again apply union bound to get that

$$P\left[\exists x \in \mathbb{X}, \left|\frac{nP_q[X = x]}{C[x]} - 1\right| \geq \alpha\right] \leq |\mathbb{X}|\frac{\delta}{\mathbb{X}} = \delta$$

□

Lemma (Probability Simplification)**.**

$$P_r[X = x] = \frac{P_q[X = x]/C[x]}{\sum_{z \in \mathbb{X}} P_q[X = z]/C[z]}$$

Proof.

$$\begin{aligned} P_r[X = x] &= \sum_{s \in \mathbb{S}:\nu(s)=x} P_r[S = s] \\ &= \sum_{s \in \mathbb{S}:\nu(s)=x} \frac{g(s)q(s)}{\sum_{s' \in \mathbb{S}} g(s')q(s')} \\ &= \frac{\sum_{s \in \mathbb{S}:\nu(s)=x} g(s)q(s)}{\sum_{s' \in \mathbb{S}} g(s')q(s')} \\ &= \frac{\sum_{s \in \mathbb{S}:\nu(s)=x} g(s)q(s)}{\sum_{z \in \mathbb{X}} \sum_{s' \in \mathbb{S}:\nu(s')=z} g(s')q(s')} \end{aligned}$$

Since we can simplify

$$\begin{aligned} \sum_{s \in \mathbb{S}:\nu(s)=x} g(s)q(s) &= \sum_{s \in \mathbb{S}:\nu(s)=x} \frac{\min_{y \in \mathbb{X}} C[y]}{C[\nu(s)]} q(s) \\ &= \frac{\min_{y \in \mathbb{X}} C[y]}{C[x]} \sum_{s \in \mathbb{S}:\nu(s)=x} q(s) \\ &= \frac{\min_{y \in \mathbb{X}} C[y]}{C[x]} P_q[X = x] \end{aligned}$$

we have that

$$\begin{aligned} P_r[X = x] &= \frac{\sum_{s \in \mathbb{S}:\nu(s)=x} g(s)q(s)}{\sum_{y \in \mathbb{X}} \sum_{s' \in \mathbb{S}:\nu(s')=y} g(s')q(s')} \\ &= \frac{\frac{\min_{y \in \mathbb{X}} C[y]}{C[x]} P_q[X = x]}{\sum_{z \in \mathbb{X}} \frac{\min_{y \in \mathbb{X}} C[y]}{C[z]} P_q[X = z]} \\ &= \frac{P_q[X = x]/C[x]}{\sum_{z \in \mathbb{X}} P_q[X = z]/C[z]} \end{aligned}$$

□

A.1.3 Efficiency Analysis

We show that the number of samples from the original distribution required to produce a sample from the homogenized distribution is $O(\frac{1}{\varepsilon})$ in expectation.

In the Salient Variable Homogenization algorithm, we have the probability of not rejecting a given sample as $g(s) = \frac{\min_x P[X=x]+\varepsilon}{P[X=X(s)]+\varepsilon}$. We know that

$$g(s) = \frac{\min_x P[X = x] + \varepsilon}{P[X = v(s)] + \varepsilon} \geq \frac{\varepsilon}{P[X = v(s)] + \varepsilon} \geq \frac{\varepsilon}{1 + \varepsilon}$$

Since each sample is independent, we can model this as a geometric distribution, and thus we have that the expected number of tries $t = \frac{1}{g(s)} \leq 1 + \frac{1}{\varepsilon}$. We thus have that in expectation, we need to sample $O(\frac{1}{\varepsilon})$ samples from the original distribution to produce one homogenized sample.

A.1.4 Empirical Effect Of Varying ε

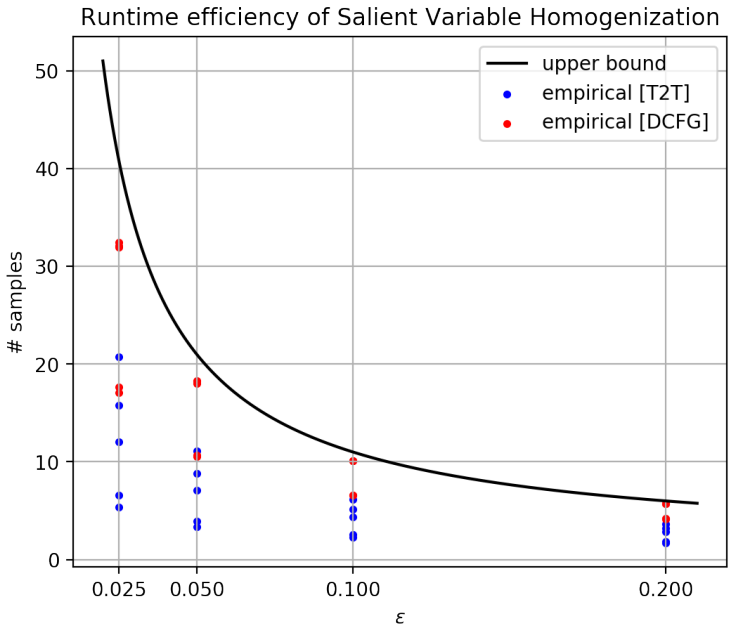


Figure A.1: Number of samples required by salient variable homogenization parameterized by an ε before a new sample is returned.

The solid line is an upper bound $\frac{\varepsilon}{1+\varepsilon}$ derived in Section A.1.3. Seen in the measured samples for different values of ε, the upper bound appropriately reflects the maximum height of the samples, with very little remaining space on the DCFG dataset and some but not much on the T2T dataset,

	Original	$\varepsilon = 0.025$	$\varepsilon = 0.050$	$\varepsilon = 0.100$	$\varepsilon = 0.200$
T2T	83.83%	+2.84pp	+1.31pp	+1.33pp	+2.45pp
DCFG	78.25%	+5.00pp	+4.50pp	+3.54pp	+3.42pp

Table A.1: Improvements in Calculator performance with homogenized datasets of various sampling parameters ε. See Section 3.5 for details on performance metrics.

and is thus a close bound. As the values of $\varepsilon \to \infty$ the bound approaches the limit 1, indicating no samples are rejected by the algorithm.

Increasing the parameter ε has the theoretical affect of causing the homogenized distribution to deviate more from uniform in its salient random variables, as shown in A.1.3. In practice, we discover that performance boosts tend to decrease with increasing ε, although the effect was not as pronounced on the T2T dataset, potentially because the unhomogenized T2T distribution is closer to uniform and thus homogenization has a limited effect for any larger ε values.

A.1.5 Empirical Evidence for Increase in Uniformity

Empirically, the Salient Variable Homogenization algorithm led to increases in uniformity in the variable being homogenized. We measure uniformity by KL divergence between the distribution being measured and the uniform distribution. A table of relative improvements is given in Table A.2.

	Length	Max Depth	Mean Depth	#Operations	#Parens
DCFG	42.98%	30.77%	27.05%	43.95%	23.63%
T2T	46.68%	30.45%	13.99%	38.82%	36.91%

Table A.2: Percentage reductions in KL-divergence from uniform of the given salient variable when homogenized at $\varepsilon = 0.025$.