Zero-Shot Semantic Parsing for Instructions

Ofer Givoli
Zero-Shot Semantic Parsing for Instructions

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Ofer Givoli

Submitted to the Senate of the Technion — Israel Institute of Technology
Tevet 5779 Haifa December 2018
This research was carried out under the supervision of Assistant Professor Roi Reichart from the Faculty of Industrial Engineering and Management, in the Faculty of Computer Science.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Roi Reichart, who has given me extraordinary freedom to follow my research interests, including changing our research problem multiple times. In our meetings, and wonderful walks, his encouragement and amazing enthusiasm greatly motivated me to push through challenges and difficulties, and generally lifted my spirit many times when I was feeling stuck. Roi has been my advisor for almost five years, and has been very generous with his time, support, ideas and advice.

The generous financial help of the Maurice and Altie Azrael Fellowship and Haifa Municipality Partial Fellowships is gratefully acknowledged.
Contents

List of Figures

List of Tables

Abstract 1

Abbreviations and Notations 3

1 Introduction 5
  1.1 Executable Semantic Parsing and λ-DCS 6
  1.2 Our Proposed Approach 7

2 Related Work 9
  2.1 Weak Supervision 9
  2.2 Semantic Parsing for Instruction Sequences 10
  2.3 Cross-Domain Settings and Zero-Shot 10
  2.4 Previous Work Related to Natural Language User Interfaces 11
  2.5 Program Synthesis 11

3 Task and Data 13
  3.1 Task 13
  3.2 Data 14
    3.2.1 Domains 14
    3.2.2 The Annotation Task 15
    3.2.3 Annotation Process and Statistics 16
  3.3 Dataset Statistics 18

4 Model 19
  4.1 Prediction 19
  4.2 Features 21
  4.3 Training 22
    4.3.1 Conditional Weight Updates (CWU) 23
    4.3.2 Two-Step Training 24
5 Experimental Results
  5.1 Experimental Setup ................................................................. 25
  5.2 Results ................................................................................. 26

6 Conclusions and Future Directions ............................................. 29

A Integration with Our Framework .............................................. 31

Hebrew Abstract ........................................................................ i
List of Figures

1.1 Initial and desired state pairs from two domains. ........................................ 6
3.1 Initial and desired state pair from the FILE domain. ................................. 17
4.1 Logical form generation during inference. .................................................... 21
A.1 The Java code for the LIGHTING domain, with the code required for integration marked with an underline. ................................................................. 32
List of Tables

3.1 The domains in our dataset. .................................................. 15

5.1 Accuracy on the test set. In parentheses: results for the in-domain setup. . . . . 27
5.2 Ablation analysis on the training set using leave-one-out cross-validation over
the domains. ........................................................................ 27
Abstract

In this work we introduce a novel task: training a parser that can parse instructions into compositional logical forms, in domains that were not seen during training. Formally, our task assumes a set of source domains, each corresponding to a toy application (e.g. a calendar or a file manager). Our goal is to train a semantic parser, using examples from the source domains, such that it could parse instructions from a different domain using nothing but the definition of the domain. We are motivated by the long term goal of allowing developers to enable a natural language user interface for their software with minimal overhead.

In our task, the annotation of the training examples does not include logical forms for the instructions. The task requires learning from weak supervision: the desired state of the application. We created a new dataset of instructions with 1,390 examples from 7 domains. Each example in the dataset is a triplet consisting of (a) the application’s initial state, (b) an instruction, to be carried out in the context of that state and (c) the state of the application after carrying out the instruction.

Our model is an extension of the parser introduced by Pasupat and Liang (2015) which was originally designed for question answering. Additional features are extracted based on description phrases that are provided for the interface methods. Also, the provided application logic is used during inference in order to dismiss some of the incorrect candidate logical forms.

We also experimented with two new training methods for this task, tailored to our zero-shot setup. The first training method is a new version of the AdaGrad algorithm which we call Conditional Weight Updates (CWU). In this training method the unconditional weight update step of AdaGrad is replaced with a partial update that depends on encountering examples from multiple domains for which the update is useful. This method did not outperform AdaGrad, which was used for the original parser.

The second training method is based on separating the training process into two steps. In the first step, weights are learned by AdaGrad using examples from a subset of the source domains. In the second step AdaGrad is used again, starting from the weights learned in the first step; and training only on examples from the source domains that were not used for the fist step. This training algorithm outperformed AdaGrad.

We hope this work will inspire readers to use our framework to collect a larger dataset and experiment with more models. Our framework is designed to allow integration with existing Java applications with minimal effort, making it easy to define new domains and collect annotated data.
Abbreviations and Notations

$s$ : State of an application
$K_s$ : Knowledge base defined by state $s$
$x$ : Natural language instruction
$z$ : Logical form
$Z_x$ : Set of candidate logical forms for an instruction $x$
$c$ : Method call
$y$ : Denotation (desired state)
$d$ : Source domain
$D$ : Set of source domains
$K$ : Number of source domains
$\phi(x, s, z)$ : Feature vector
$\theta$ : Weight vector
Chapter 1

Introduction

The idea of interacting with machines via natural language instructions and queries has fascinated researchers for decades. Early attempts include the famous SHRDLU program presented by [Win71], which could parse instructions from a single narrow domain: handling toy blocks on a table. In recent years industry giants took the challenge of developing natural language interface platforms known as “intelligent personal assistants”. Such assistants include Alexa (Amazon), Google Assistant, Siri (Apple), and Cortana (Microsoft).

In the near future, we may find ourselves in a world where most new devices and software can be, at least partially, operated via a natural language user interface (NLUI). If so, we better seek answers to the following questions: How general can we make the process of developing a NLUI for a given application? Will every developing team need to hire NLP experts to develop a NLUI for their specific application? Can we hope for a general framework that once trained on annotated data from a set of domains, does not require annotated data from a newly presented domain? Previous work on tasks related to NLUI for applications mostly relied on in-domain data (e.g. [AZ13, LPL16]), and papers that did not rely on in-domain data did not attempt to parse instructions into compositional logical forms (e.g. [KRS16]).

In this work we introduce a novel task: training a parser that can parse instructions into compositional logical forms, where the instructions are from domains that were not seen during training. Formally, our task assumes a set \( D = \{d_1, ..., d_n\} \) of source domains, each corresponding to a toy application (e.g. a calendar or a file manager), with simple data structures and an API consisting of a few interface methods. Our goal is to develop a semantic parser that trains on examples from a set of source domains \( D \) and can then parse instructions from a new target domain, without getting any training examples from that domain. Instructions are parsed into logical forms that represent a method call with specific arguments.

The annotation of the training examples does not include logical forms. Therefor, the task requires learning from a weak supervision: the desired state of the application. This is in accordance with the learning paradigm presented by [CGCR10] and followed by many semantic parsing papers (e.g. [BCFL13, PL15]).

As part of presenting the task, we created a new dataset of instructions with 1,390 examples from 7 domains. Each example in the dataset is a triplet consisting of (a) the application’s
(a) Container Management System:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Desired State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction: "remove the longest container"

(b) Lighting Control System:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Desired State</th>
</tr>
</thead>
<tbody>
<tr>
<td>floor 1</td>
<td>floor 2</td>
</tr>
<tr>
<td>hall</td>
<td>hall</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Desired State</th>
</tr>
</thead>
<tbody>
<tr>
<td>floor 1</td>
<td>floor 2</td>
</tr>
<tr>
<td>living room</td>
<td>living room</td>
</tr>
</tbody>
</table>

Instruction: "turn off the light in the bedroom on floor 2"

Figure 1.1: Initial and desired state pairs from two domains. The instructions order a transition from an initial state to a desired state. The initial state, visualized in the left side of each of the above tables, is presented to the workers as the "current state" they are in. The desired state, visualized on the right side, is presented as the state they want to arrive at. See section 3.2.1 for a description of the domains. (a) Shipping containers can be either empty (sketched as gray) or full (sketched as blue), and with varying length. (b) Rooms with the lights on/off are sketched with orange/gray background respectively.

initial state, (b) an instruction, to be carried out in context of that state, and (c) the state of the application after carrying out the instruction, also referred to as the desired state. Figure 1.1 demonstrates examples from two of the domains in our dataset. We achieved an averaged accuracy of 44.5% on the test set, compared to 39.1% which was yielded by the parser when trained with its original AdaGrad training algorithm [DHS11].

1.1 Executable Semantic Parsing and λ-DCS

Semantic parsing is the task of mapping natural language into formal meaning representation. In executable semantic parsing the formal meaning representation can be executed, often in the context of some knowledge base, and the results of this execution is called denotation.

The space of logical forms that our parser maps to is based on a formal language called Lambda Dependency-Based Compositional Semantics, or λ-DCS, which is presented in [Lia13]. λ-DCS is a restricted form of λ-calculus that allows a more compact logical form representation by eliminating variables and using implicit existential quantification.

The logical forms in λ-DCS are defined recursively, where every logical form is either a unary denoting a set of values, or a binary denoting a set of value pairs. The primitive binaries are the relations of the knowledge base. Let \( z_1 \), \( z_2 \) be unaries, and let \( r \) be a binary. The original operators of λ-DCS that combine logical forms into larger ones and are relevant to this work are the following:

1. Intersection: \( z_1 \cap z_2 \)
2. Union: \( z_1 \cup z_2 \)
3. Join: $r.z_1$

4. Reverse: $R[r]$, which denotes the binary: \[ \{(e_1, e_2) : (e_2, e_1) \in r\}. \]

5. Superlatives: $argmax(z_1, r)$, which denotes the unary: \[ \{e : (e, m) \in r\} \text{ where } m := \max\{m : (e, m) \in r\}; \text{ and similarly } argmin(z_1, r). \]

6. Lambda (e.g. $\lambda x[r.x]$)

Suppose we are given a knowledge base about some set of containers, and the knowledge base contains a relation $\text{length}$ that defines the length of the the containers, and the relation $\text{type}$ that defines the type of entities. A semantic parser can be used to map the natural language phrase “the longest container” into the following logical form:

$$\text{argmax}(R[\text{type}].\text{ShippingContainer}, R[\text{length}])$$

This logical form can be executed as a query over the knowledge base and the result (referred to as the denotation of the logical form) will be the set of container entities which have the maximum length.

### 1.2 Our Proposed Approach

Our model is an extension of the floating parser introduced by [PL15], which was designed for a question answering task. We experimented with two new training methods for this task, tailored to our zero-shot setup. In the first training method we present a new version of AdaGrad [DHS11], the stochastic gradient descent algorithm used in [PL15], which we call Conditional Weight Updates (CWU). In this new training method the unconditional weight update step of AdaGrad is replaced with a partial update that depends on encountering examples from multiple domains for which the update is useful. This method did not outperform AdaGrad.

The second training method is based on separating the training process into two steps. In the first step, weights are learned via AdaGrad. The training examples used are from a subset of the source domains. In the second step, AdaGrad is used again with examples from different source domains, starting from the weights learned in the first step. This training method yielded an absolute increase of 5.4% in average accuracy relative to AdaGrad, from 39.1% to 44.5%.

We extract additional features based on co-occurrence of primitive logical forms and description phrases provided for each interface method. Also, we use the provided application logic to dismiss candidate logical forms that represents a method call which does not modify the application state or results in an exception being thrown from the application logic.
Chapter 2

Related Work

A lot of work has been done on executable semantic parsing, and it can be classified as either work on question answering (e.g. [CGCR10, PL15]) or instruction parsing (e.g. [AZ13, LPL16]). The result of executing a logical form is either an answer or a change in some state, respectively.

Our work is the first to address the novel task of parsing natural language instructions into compositional logical forms in zero-shot settings. We shall now discuss the fields that intersect with our task and the relevant previous work. We begin with a discussion on previous work in which semantic parsers are trained with weak supervision. We then discuss semantic parsing for instructions and semantic parsing in cross-domain and zero-shot settings. Finally, we discuss previous work related to natural language user interfaces.

2.1 Weak Supervision

Since [CGCR10], a lot of papers presented parsers that are trained with weak supervision [KM12, BCFL13, KCAZ13, AZ13, BL14, WBL15, PL15, LPL16], where the logical form of the question/instruction is not part of the annotation. Instead, the annotation of each question or instruction consists of an answer or outcome, respectively. [GRCR11] presented a more extreme approach, completely unsupervised and driven by confidence estimation using unannotated in-domain utterances.

Older work is based on a more direct (and expensive) supervision: annotation that consists of the logical form of the utterance (e.g. [ZC05, KZGS10]). Datasets that include such annotation, such as the Geoquery dataset [ZM96, TM01] and the Free917 dataset [CY13], were expensive to construct and were hence usually small. [WBL15] introduced a new approach for constructing such datasets, in which the human workers paraphrase automatically generated utterances. In this work training is done via weak supervision: the outcome of the instruction.
2.2 Semantic Parsing for Instruction Sequences

While the zero-shot setup is the aspect that makes our task challenging, other tasks for semantically parsing instructions are challenging due to the length of the natural language text being parsed. [MSK06] introduced a classic task in semantic parsing, where the goal is to parse navigation instructions for an agent in a given map. Unlike our task, where a single sentence instruction is parsed, their task requires parsing a sequence of sentences (4.6 sentences per sequence on average). Another important difference is that their task consists of a single domain, and provides in-domain training data.

[LPL16] introduced a new task for semantically parsing instructions over three small domains. The task requires parsing a sequence of five sentences, each requesting a change in the world state. Their task does not include a zero-shot setup, rather, they train a parser for each domain separately with in-domain data.

2.3 Cross-Domain Settings and Zero-Shot

A lot of the proposed frameworks are designed to train a parser using in-domain data [KM12, BCFL13, AZ13, BL14, WBL15, LPL16], meaning that the training examples and test examples are from the same domain. [CY13] and [KCAZ13] introduced semantic parsers for question answering that can parse utterances from Free917 ([CY13]) such that no Freebase entities or relations appear in both training and test examples.

Among the papers that rely on in-domain training data, many rely on a domain-specific lexicon [KZGS10, GN11, KM12, ZC05, CY13]. The lexicon maps natural language phrases to logical symbols (e.g. a primitive logical form representing an entity from the knowledge base). The common approach is to automatically construct the domain-specific lexicon using some additional domain-specific resource that is associated with the entities and relations of the given domain. Such a resource can be either a very large corpus ([GN11, KM12]), search results from the world wide web ([CY13]) or the logical form annotation of sentences from the training set ([ZC05, KZGS10]). In our task, none of the above resources is available for the target domain, but rather only the formal definition of the target domain is available.

In [PL15], the authors introduced a new dataset of questions about Wikipedia tables, such that the context of each question is a single table. The evaluation is done on questions about tables that are not seen during training, but there are table columns across the train/test split that share column headers (which correspond to primitive logical forms that represent relations). Our parser is based on the floating parser introduced in that paper, and the space of logical forms we use is very similar to theirs (see chapter 4).

[HB17] and [SY17] experimented with the Overnight dataset ([WBL15]) in cross-domain settings, using training data from the non-target domains. These paper did not experiment with zero-shot setup, and they both observed that the less in-domain training data was used, the more training data from other domains was valuable.

Recently [HB18] presented a new task and a framework for zero-shot semantic parsing.
for question answering, over the Overnight dataset. Their framework, which requires logical form annotation, decouples the parsing into two sub-tasks: (a) mapping natural language into domain-independent logical forms that contain slots instead of KB constants; and (b) replacing the slots with KB constants.

2.4 Previous Work Related to Natural Language User Interfaces

For some NLUI setups, it is not required to map natural language to a meaning representation over a space of compositional logical forms. If the problem is limited to natural language utterances with a low compositionality then other methods such as information extraction or shallow semantic parsing might suffice. For example, [YBVD14] showed that questions from the WebQuestions dataset presented in [BCFL13] can be answered using information extraction.

In slot tagging, rather than constructing compositional logical forms, phrases in the utterances are tagged as task-specific arguments (e.g. time, location) called slots. For example, [KRS16] presented such a framework for a dataset for slot tagging related to the Microsoft Cortana intelligent personal assistant. Their dataset contain 16 domains (e.g. ALARM and CALENDAR), and most slots of each domain appear in multiple domains (e.g. time, location). Their system can handle instructions from domains that are not seen during training. [CRX+17] created a dataset for instruction parsing for 45 domains (APIs in the Thingpedia repository), based on the paraphrasing approach presented in [WBL15], and they did not experiment with a zero-shot setup. [FMMD17] and [PCKS18] experimented with parsing instruction in multiple domains and used transfer learning, always using in-domain data as well as training data from domains other than the target domain. Similarly to the cross-domain papers on Overnight mentioned above, both these papers observed that the less in-domain training data was used, the more training data from other domains was valuable.

Finally, a lot of work has been done in the field of goal-oriented dialog (e.g. [GMRB+17]) in which a human user interacts with a system by multiple exchanges of questions from the system and instruction/answer utterances from the user. The intent of the user is represented by slot tags over the user’s utterances, which have low compositionality. A lot of these papers experimented with multiple domains, but not in a zero-shot setup. Conversely, the recent paper [ZE18] experiments with dialog generation in a zero-shot setup.

2.5 Program Synthesis

Some relevant work has also been done in program synthesis. In [RGMF15, DGH+16] the proposed frameworks take as input a domain-specific language (DSL) definition and training data which is pairs of English descriptions and corresponding programs. A synthesizer, that given a new utterance in that domain synthesizes DSL code, is constructed. The given DSL is assumed to have a set of operations that are similar to concepts that the human users might express in natural language, which is similar to our motivation. [LWZE18] presented a new task for mapping natural language to bash code. All these tasks do not involve zero-shot parsing,
and the proposed frameworks rely on in-domain training examples. Also, unlike our task, their utterances are not parsed in the context of an application state.
Chapter 3

Task and Data

In this section we describe our task and the dataset we collected. We start with an overview of the task, and provide definitions and notations. We continue with a discussion about the dataset: the domains, the construction process and finally some statistics.

3.1 Task

Our task involves parsing instructions into compositional logical forms, in toy application domains. For example, one of our domains is LIGHTING, in which a simple lighting control system controls the lights in a house, turning the lights on and off in rooms in response to the user’s instruction.

Formally, a domain has a set of interface methods (e.g. turnLightOn and turnLightOff) that can be invoked with some arguments. Each argument is a set of entities. In the example of LIGHTING, the argument is a set of Room entities. There are two kinds of entity types: domain-specific (e.g. Room) and non domain-specific (Integer, String).

A state defines a knowledge base which contains a set of \((e_1, r, e_2)\) triplets, where \(e_1\) and \(e_2\) are entities and \(r\) is a relation. For example, a state in the LIGHTING domain might contain the triplet \((room3, floor, 2)\) which indicates that room3 is on the second floor of the house. See figure 1.1 (b) for a visualization of two possible states in the LIGHTING domain. In the first one, there is a bedroom on the second floor with lights turned on. If that room is represented in the knowledge base of that state by the entity room1, the following triplets will be in the knowledge base: \((room1, name, bedroom)\), \((room1, floor, 2)\) and \((room1, lightMode, ON)\).

Our task is limited to mapping an utterance into a single method call. A method call formally consists of an [interface method, argument list] pair. An argument can be either a primitive entity, a non-primitive entity or a set thereof. A method call changes the application state according to the provided application logic. This work is limited to domains in which the application state can be represented by a finite set of (entity,relation,entity) triplets. Also, this work does not support domains that require to identify a phrase, that appears in the utterance but does not match any label in the knowledge base, as an argument to an interface method (e.g. the name of a new entity to be created).
Our dataset, described in the next subsection, contains examples in multiple domains. Each example is an \((s, x, s')\) triplet, where \(s\) is an initial application state, \(x\) is a natural language instruction and \(s'\) is a desired application state resulting from carrying out the instruction \(x\) on the state \(s\).

We are motivated by the goal of training a parser once with a given set of domains \(\{d_1, \ldots, d_n\}\), and then employing this parser in the context of new, previously unseen, domains \(d_{n+1}, \ldots\) without any additional training.

### 3.2 Data

Experimenting with this task requires a dataset of instructions from multiple domains. With such a dataset, one can train a parser on examples from some of the domains, and then evaluate on the remaining domains. Each example in our dataset consists of an initial state, an instruction and the resulting state (a.k.a. desired state).

We did not annotate the instructions with logical forms, following [CGCR10, LJK11, GCR11] and a lot of the work in semantic parsing since. Rather, the annotation of each example consists only of the desired state (i.e. the result of performing the instruction on the initial state), and candidate logical forms are created by the parser as latent variables.

A common practice for constructing a NLUI instruction dataset is presenting human workers with some visualization of pairs consisting of initial and desired states. The workers are asked to write an instruction that can be executed in order to transfer the system from the initial state into the desired state (see, e.g. [LPL16]). Following this approach, we have collected a dataset of 1,390 examples from 7 domains, summarized in table 3.1.

#### 3.2.1 Domains

We collected examples from 7 domains:

**CALENDAR** Removing calendar events and setting their color.

**CONTAINER** Container management system: loading, unloading and removing shipping containers.

**FILE** File manager: removing files and moving them from one directory to another.

**LIGHTING** Lighting control system: turning lights on and off in rooms inside a house.

**LIST** Managing a simple list: removing elements and moving an element to the beginning/end of a list. The motivation comes from the need to manage task lists, shopping lists, etc. For the sake of simplicity our domain deals with lists of integers.

**MESSENGER** Creating/deleting chat groups and muting/unmuting them.

**WORKFORCE** Workforce management system: assigning employees to a new manager, firing employees, assigning an employee to a new position and updating an employee’s salary.
Table 3.1: The domains in our dataset. The interface method parameters are presented in Java syntax. In the second column, the properties of the non-primitive entities appear in parentheses.

We were motivated to choose domains that allow instructions with interesting linguistic phenomena such as superlatives (e.g. “remove the longest container” in the CONTAINER domain, see figure 1.1 (a)), spatial language (e.g. “turn off the light in the bedroom on floor 2” in the LIGHTING domain, see figure 1.1 (b)), temporal language (e.g. “delete my last two appointments on Thursday”, in the CALENDAR domain). Also, the domains were chosen to be rich enough to allow utterances with highly compositional logical forms (see section 3.2.3). More details on each domain, including non-primitive entity types and interface methods are presented in table 3.1.

3.2.2 The Annotation Task

We provided workers with state pairs of the form: (initial state, desired state), see figure 1.1. The construction process of these pairs was similar to the one in [LPL16]. Given a domain and an interface method, we randomly generated a state pair with the following steps:

1. Randomly generating an initial state. For example, in the LIGHTING domain (see figure 1.1 (b)), we randomly select the number of floors, number of rooms in each floor, and for each room we randomly select a name (e.g. bedroom) from a list of possible names, and a light mode (either ON or OFF).

2. Randomly selecting arguments for the interface method. For example, in the LIGHTING domain we randomly select a set of rooms as an argument for the interface method (turnLightOn or turnLightOff).

3. Executing the interface method with the selected arguments on the initial state. If the result is a state that is identical to the initial state, or if an error occurred during execution, we go back to step 2. After 1,000 failed attempts we deem the random initial state as problematic and go back to step 1.
We intend to release the code that generated those state pairs, and it can be used to easily extend our dataset by defining additional domains and increasing the number of examples per domain.

3.2.3 Annotation Process and Statistics

We wanted the human workers to provide us with the same utterance they would naturally speak to an intelligent personal assistant in order to reach the desired state. We also wanted our dataset to contain instructions with highly compositional logical forms. MTurk workers are getting paid per instruction and so they are incentivised to write instructions that are quick to type. This preference tends to result in low compositionality, challenging us to find a way for collecting a dataset with higher logical compositionality.

Comparison with Previous Work  Different approaches exist for encouraging workers to provide utterances with highly compositional logical forms. For example, in [PL15], where workers were asked to come up with questions about Wikipedia tables, for each question one of 36 generic prompts was shown, such as “The question should require calculation” or “contains the word 'first' or its synonym”. While this is an effective way to get natural language utterances with deeper compositionality, it comes with the cost of sacrificing to some degree the authenticity of the utterances.

[WBL15] took a different approach in which synthetic utterances are automatically generated from a domain specific grammar, and the role of human workers is limited to paraphrasing the synthetic utterances. This approach therefore does not allow the human workers to freely come up with a natural language utterance on their own. Note that in [WBL15] only a single random initial state was generated per domain, while we generate a random initial state per example.

We thus used a different approach for encouraging deeper compositionality that, even if less effective, does not pose strict limitations on the utterances that the human workers provide. We follow [LPL16] which also collected a dataset of instructions. Unlike our task in which every example contains a single utterance describing a single method call, in their task each example contained a text with 5 utterances describing 5 state transformations (“actions”). They found it nontrivial to obtain interesting linguistic context-dependence, which was important for their task. Consequently, they modified their domains in order to get utterances that are more context-dependent.

For example, one of their domains involves people standing on a one-dimensional stage, each wearing a shirt and a hat with varying colors, and the transformations between states are the result of people entering/leaving the stage, moving along the stage and exchanging hats with each other. They modified this domain by removing any visual indication of peoples’ absolute position on the stage, and limited the state transformations involving people moving along the stage such that a person always moves next to another person. The goal was to have the workers provide phrases like “to the left of the man in the red hat” rather than “to position 2”.

16
Figure 3.1: Initial and desired state pair from the FILE domain. Note the file names with a the 4-digit suffix, incentivising workers to avoid typing file names, and instead coming up with utterances referring to other properties, hopefully resulting in deeper compositionality. For this state pair the worker provided the utterance: “Delete the largest file”.

Our Process  Similarly to [LPL16], we tried to define domains that would result in examples with deeper logical compositionality. We leveraged the fact that the MTurk workers are getting paid per utterance and are thus incentivised to write an instruction which is quick to type. We tried to define our domains in a way that would correlate shallow compositionality with longer typing time. For example, in the FILE domain, file names include a 4 digit suffix in order to incentivise workers to avoid typing an utterance with the file name (e.g. “delete the file car0420”), and prefer other ways to describe the file (e.g. “delete the largest file”). See figure 3.1 for a concrete example.

Another method we used is automatically filtering out, to some extent, random (initial state, method call) pairs that do not allow workers to query entities using superlatives/comparatives. The criterion is the existence of a scalar property (e.g. Integer) which distinguishes the argument entities in a way that can be described by a superlative/comparative statement. For example, in figure 1.1 (a) the state pair allows the worker to use a superlative in order to describe the relevant entity, and so they did (i.e. using the phrase “the longest container”). This is similar to the way [LPL16] restricted actions as described above.
3.3 Dataset Statistics

We employed the MTurk platform and recruited workers located in the US with at least 1,000 approved tasks\(^1\) and a task approval rate of at least 95%. Workers were paid 15-23 cents per task (i.e. per utterance they write given a state pair). Each state pair was given to only one worker. The average instruction length in the training set is 8.1 words.

We analyzed 200 examples from the training set, sampling 10 examples from each interface methods. From these examples, 16.5% had an incorrect utterance, among them in 27% of the examples the workers provided an utterance that refers to meaningless details in the sketch rather than to the state it represented. For example, a worker provided the utterance: “move red highlighted files into the work directory”. The files were highlighted in the provided sketches only to help the workers identify the difference between the two sketches quickly, and they were instructed not to refer to such details in the utterance they provide. The dataset contains utterances with temporal phrases (e.g. “delete all appointments after the 9:00 meeting on the 9th.”, from the CALENDAR domain) and spatial phrases (“remove the topmost 2 containers and the bottom container” from the CONTAINER domain).

\(^1\)Called “HITs” in Amazon Mechanical Turk.
Chapter 4

Model

In this chapter we describe our approach for solving the task. We begin by describing the prediction process: modelling the probabilities of candidate logical forms, defining the space of logical forms, and the parsing process, i.e. how the set of candidate logical forms is generated given an initial state and an instruction utterance; all this using the parser presented in [PL15] (which was designed for question answering) with our minimal modifications to support instruction parsing.

We then discuss the features used, the ones taken from [PL15] and the ones we added. Finally we introduce two training approaches tailored for our task.

4.1 Prediction

Our zero-shot parser is based on the floating parser from [PL15]. In this section we briefly describe their parser (which was designed for question answering), with our minimal modifications to support instruction parsing.

For each inference, the input of the parser is an initial application state $s$, a set of interface methods and their description phrases, and a natural language instruction $x$. The state $s$ defines a knowledge base $K_s$ of $(entity, relation, entity)$ triples. The parser generates a set of logical form candidates $Z_x$ that can be executed over the state $s$ to produce a method call $c$ formulated as an [interface method, argument list] pair. The method call $c$ can be invoked in the context of state $s$ with the provided application logic, producing the denotation $y = [z]_s$ which is the resulting state.

For each logical form $z \in Z_x$ the parser extracts a feature vector $\phi(x, s, z)$. In section 4.2 we discuss the original feature set of the parser as well as our extension that is designed to deal with zero-shot parsing. The probability of a logical form candidate $z \in Z_x$ is defined by a log-linear model:

$$p_\theta(z|x, s) = \frac{\exp(\theta^T \phi(x, s, z))}{\sum_{z' \in Z_x} \exp(\theta^T \phi(x, s, z'))}$$  \hspace{1cm} (4.1)$$

where $\theta$ is the weight vector. The logical form with maximal probability is chosen as the predicted logical form, and its denoted state is the predicted desired state.
Logical Forms Following [PL15], our parser maps utterances to logical forms over a variant of λ-DCS. In section 1.1 we described the relevant operators of λ-DCS that combine logical forms to a larger logical form. We introduce a small extension to the original λ-DCS language to support logical forms that represent a method call. Suppose \( z_1, \ldots, z_n \) are logical forms denoting a set of entities, and suppose \( f \) is a primitive logical form that denotes an interface method. Then the logical form \( f(z_1, \ldots, z_n) \) denotes a method call in which \( z_1, \ldots, z_n \) are the arguments of the interface method \( f \).

Generating the Set of Candidate Logical Forms The floating parser is a bottom-up beam-search parser, in which a dynamic programming table is being filled with derivations. Each cell in the table corresponds to a derivation size (defined as the number of rules applied) and a logical form category. Unlike more rigid chart parsers, the cells do not correspond to specific token spans. Accordingly, the rules for combining logical forms do not depend on a mapping from the logical forms to token spans, and therefore the authors refer to those rules as “floating rules”.

The intractable number of possible logical forms is dealt with by limiting the beam size, i.e. the number of logical forms with highest assigned probability that are kept in each cell. There is also a pruning step that removes partial logical form that are invalid or useless. For example: partial logical forms that violate the typing system (e.g. \( \text{bedroom} \cup 2 \)), denote an empty set or apply a superlative operation to a singleton.

The cells of the table being filled are all the cells up to the decided maximal logical form size. The set of candidate logical forms are then the ones in all cells for the category ROOT, which in this work is a category indicating that the logical form represents a method call. Figure 4.1 illustrates a full derivation of a candidate logical form.

Primitive logical forms that represent relations, operations (e.g. \( \text{argmax} \)) and interface methods (e.g. \( \text{turnLightOn} \)) are created automatically at the beginning of the parsing process. In contrast, primitive logical forms that represent entities must be anchored to a matching phrase in the utterance. For example, the logical form \( \text{bedroom} \) (denoting the primitive entity \( \text{bedroom} \), which is a room’s name) can only be generated by a lexical rule that matches “bedroom” with some phrase in the utterance. The motivation for this is that primitive entities tend to appear explicitly in the utterance, while relations, logical operators and interface methods can be expressed via unexpected language, or might not explicitly appear at all (e.g. in the utterance “turn off the light in the bedroom” the \( \text{roomname} \) relation itself does not explicitly appear).

See [PL15] for more details and the full list of rules. We added an additional derivation rule that derives the logical form \( f(z_1, \ldots, z_n) \) given the primitive logical form \( f \) (denoting an interface method) and the logical forms \( z_1, \ldots, z_n \) (each denoting a set of entities).

We added the following logic, that filters some of the incorrect candidate logical forms: the provided application logic is used to dismiss candidate logical forms when they represent a method call \( c \) which does not modify the state or results in an execution error. Meaning, \( c \) is invoked on the initial state \( s \) and if the result is a state which is identical to \( s \), or if an exception has been thrown by the application logic, we dismiss the candidate logical form. For example,
Figure 4.1: Logical form generation during inference. Logical forms are shown in rectangles. Solid lines show derivations via anchored rules while dotted lines show derivation via unanchored (floating) rules. The logical forms turn off (denoting an interface method) and roomname and floor (denoting relations) are derived via the floating rules $\emptyset \rightarrow \text{Method}$ and $\emptyset \rightarrow \text{Relation}$ (not illustrated here).

in the LIGHTING domain the lights in rooms can be turned on and off, and if $c$ is a method call that turns off the lights in rooms in which the lights are already off, the state will not change. In the WORKFORCE domain, in which employees can be fired, firing an employee who is currently the manager of other employees results in an exception being thrown.

In our task, due to the need to parse utterances from unseen domains, there is a lot of uncertainty regarding the correct way to parse the utterance. Therefore for this work we chose to extend the floating parser, rather than parsers with more rigid derivation rules such as CCG parsers.

4.2 Features

Given a state $s$, an instruction $x$ and a logical form $z$, we extract a vector of features $\phi(x, s, z)$. All the features described in this section are from [PL15] unless specified otherwise. Most of the features are of the form $(f(x), g(z))$ where $f$ and $g$ extract some information such as the identity of a primitive logical form in $z$ and POS tags of tokens in $x$. The following features are extracted:

1. Phrase-predicate co-occurrence features: extracted according to co-occurrence between n-grams from $x$ and primitive logical forms from $z$. We use the term “predicate” to be consistent with the terminology used by [PL15], but notice that in our model all this also applies to primitive logical forms denoting interface methods.

There are two types of these features:

(a) Lexical features: features that indicate the appearance of a specific phrase in the utterance.
(b) Non-lexical features: features that are extracted whenever there is a string matching between a token span in \( x \) and a primitive logical form in \( z \) (e.g. \{“bedroom”,bedroom\}).

2. Missing-predicate features: features that indicate that a predicate (e.g. bedroom) does not appear in \( z \) even though a matching phrase (e.g. “bed room”) appears in the utterance. These are non-lexical features.

3. Description phrase features: Description phrases are phrases that the application builder\(^1\) provides for the interface methods, and they act as alternative labels for the primitive logical form representing the interface method. For example, the interface method removeEvents from the Calendar domain has the description phrases remove and cancel. Each interface method in our dataset has 1-3 description phrases. We use the description phrases to extract additional phrase-predicate co-occurrence features and missing-predicate features as described above. For example, consider the utterance “Delete the largest file” from the File domain, with the logical form:

\[
\text{removeFiles} \left( \text{argmax}(R[type].\text{File}, R[sizeInBytes]) \right)
\]

The interface method removeFiles has the description phrase delete, which matches the phrase Delete in the utterance, resulting in the extraction of the corresponding co-occurrence features. Conversely, when the parser considers logical forms that contain the method moveFiles instead of removeFiles, it will extract the corresponding missing-predicate features, because while a match between a primitive logical form and the unigram “Delete” is possible, it does not occur in the considered candidate logical form.

4. Logical form size features: We added features that correspond to the size of a candidate logical form (the number of derivation rules applied). The extracted features indicate that the logical form size is larger than \( n \), for any \( n \geq 2 \). These features capture a domain-independent preference for simplicity.

4.3 Training

Given a set of training examples from the domains \( D = \{d_1, ..., d_n\} \), we wish to learn a weight vector \( \theta \) that will yield good results for the unseen domain \( d_{n+1} \). We shall first describe the objective function used by [BCFL13] and [PL15], and then we will introduce two training algorithms tailored to our task.

Given a state \( s \), an utterance \( x \), and the the set of candidate logical forms \( Z_x \) constructed by the parser, the probability assigned to a denotation \( y \) is the sum of probabilities assigned to all the candidate logical forms with that denotation:

\[
p_\theta(y|x, s) = \sum_{z \in Z_x : [z]_s = y} p_\theta(z|x, s)
\]

\(^1\)The builder is the one providing the formal definition of an applications, following the terminology in [WBL15].
where $p_\theta(z|x,s)$ is defined by equation 4.1. The objective is the L1 regularized log-likelihood of the correct denotations in all the training data:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i|x_i,s_i) - \lambda \|\theta\|_1$$

[BCFL13, PL15] optimized this objective using AdaGrad (introduced in [DHS11]) - a stochastic gradient descent algorithm with per-feature learning rate, which we use as well.

In our task we train a parser on the domains $d_1, ..., d_n$ and then evaluate it on the domain $d_{n+1}$. Let us denote with $\theta^*_{[1,n]}$ the optimal weights for the training domains. Since the problem is not convex we can not hope to actually find $\theta^*_{[1,n]}$, but training with classic AdaGrad on the training domains is an effort to approximate $\theta^*_{[1,n]}$ as best as possible. And indeed, that is our baseline. In this section we shall discuss how we can do better than that.

For the sake of discussion, let us imagine we randomly sampled our $d_1, ..., d_{n+1}$ domains from an infinite population of domains. Let us denote with $\theta^*$ an optimal weight vector with regards to the objective over that infinite domain population. Thus, when we are given $d_1, ..., d_n$ domains to train on, our real goal is to learn weights as close to $\theta^*$ as possible (rather than $\theta^*_{[1,n]}$) because that would maximize the expected accuracy on the unseen domain $d_{n+1}$. The question is how, using only examples from $d_1, ..., d_n$, do we get as close to $\theta^*$ as possible. We shall now discuss our two proposed solutions to this challenge: one is based on training the weights in two steps, each with different domains; and the other involves modifying AdaGrad such that updates to weights would depend on evidence that the update will be beneficial for multiple domains.

### 4.3.1 Conditional Weight Updates (CWU)

In this approach the way AdaGrad updates weights is modified. The motivation is to partially suppresses a weight update until there is evidence suggesting the update is beneficial for at least $M$ domains (where $M$ is a hyper-parameter). The guiding intuition was that conditional weight updates, biased in favor of updates that benefit multiple domains, are likely to steer $\theta$ more towards $\theta^*$ than the original AdaGrad which lacks this bias.

Let us now describe the algorithm, formulated in Algorithm 1 which is called by the modified AdaGrad whenever a weight is to be updated. Throughout the training process, whenever a weight update is supposed to occur for weight $\theta_j$, after parsing an example from domain $d_i$, we only carry out the fraction $\alpha \in (0, 1]$ of the update, where $\alpha$ is a hyper-parameter.

The rest of the update is considered “pending” and is added to the map $\text{PendingWeights}(d_i,j)$, abbreviated $PW$. This map is maintained throughout the training process, holding the pending update value of each feature $j$ and domain $d_i$. As soon as there are pending update values, with the same sign, for a feature $j$ in at least $M$ domains, the one with the smallest absolute value is added to the weight $\theta_j$. Also, that quantity is then removed from the corresponding cells in $PW$. 
Algorithm 1: Conditional Weight Updates (CWU): the modified AdaGrad weight update

**Input:** Domain: \( d_i \)

- Feature to update: \( j \)
- Update value: \( u \)

\[
\theta_j \leftarrow \theta_j + \alpha \cdot u \\
PW(d_i, j) \leftarrow PW(d_i, j) + (1 - \alpha) \cdot u \\
K \leftarrow \{ k : sign(PW(d_k, j)) = sign(PW(d_i, j)) \} \\
\text{If } |K| < M \text{ then stop} \\
k' \leftarrow \arg\min_{k \in K} |PW(d_k, j)| \\
u' \leftarrow PW(d_k', j) \\
\theta_j \leftarrow \theta_j + u'
\]

For each \( k \in K \) do:

\[
PW(d_k, j) \leftarrow PW(d_k, j) - u'
\]

4.3.2 Two-Step Training

Our second training algorithm, TWOSTEP, is based on learning weights in two steps. We denote the set of domains on which we train the parser as \( D = D_1 \cup D_2 \), where \( D_1 \) and \( D_2 \) are some partition of \( D \).

Now let us describe the training algorithm, formulated as Algorithm 2. First, AdaGrad is used to learn weights by training only on examples from the domain subset \( D_1 \). In the second step, training is done with the second subset of domains \( D_2 \), starting from the weights that were learned in the first step.

Since the objective being optimized is not convex, the initial weights of the second step matter. The motivation is allowing the second step to get closer to \( \theta^* \) by starting from weights that are optimized for different domains than the ones currently being trained on; hopefully benefiting from the supervision signal received for candidate logical forms that were constructed with the weights optimized for the other domains.

Algorithm 2: Two-step training

**Input:** The domain partition: \( \{ D_1, D_2 \} \)

**Output:** Weight vector: \( \theta_2 \)

\[
\theta_1 \leftarrow \text{AdaGrad}(D_1, \theta_0) \\
\theta_2 \leftarrow \text{AdaGrad}(D_2, \theta_1) \\
\text{return} \ (\theta_2)
\]
Chapter 5

Experimental Results

5.1 Experimental Setup

The Task  Given the interface of a target domain, an initial state and an instruction, our task is to construct a method call that matches the instruction in the context of the initial state. Our goal is to train a compositional semantic parser that can parse instructions from unseen domains. Therefore, we evaluate our parser in zero-shot settings: when evaluating the parser on domain \( d \) (i.e. when \( d \) is the target domain), we train and tune hyper-parameters using the training data from other domains and then evaluate on examples from \( d \).

The dataset we used is described in chapter 3. It contains 7 domains, and on average 198.6 examples per domain which were randomly split per example category into training and test sets with a ratio of 1:1, removing examples in which the utterance contained more than one sentence.

Hyper-Parameter Tuning  The hyper-parameters were tuned using leave-one-out cross-validation over the source domains. Meaning, if our source domains are \( d_1, \ldots, d_K \), we tune the hyper-parameters by training using examples from each \( K - 1 \) subset, while using the training data of the remaining source domain as the validation set. This way, we do not use in-domain data for hyper-parameter tuning. The metric optimized is the average accuracy over all the held-out source domains.

For the TwoSTEP method, we consider as a hyper-parameter the order of the domains in \( D \), assigning the first \( M \) domains to \( D_1 \) and the rest to \( D_2 \) (the held-out source domain is excluded from the ordered list), where \( M \) is another hyper-parameter. When a final parser for the target domain is trained, we increase by one the size of either \( D_1 \) or \( D_2 \), whichever is larger, because this way the ratio between the size of \( D_1 \) and \( D_2 \) is kept as similar as possible to the ratio during the hyper-parameter tuning.

The values used for the grid search follow. The L1 regularization coefficient: \{0.001, 0.01\}; the initial step-size: \{0.01, 0.1\}; the number of training iterations (for the TwoSTEP method, the number of training iterations in the second step): \{1, 2, 3\}. For the TwoSTEP method only: the number of training domains used for the first step (\( M \)): \{3, 4\}; the number of training iterations
during the first step: \{2, 4\}; also, one of three random domain orderings for determining the $D_1, D_2$ partition was used. For the CWU method only: the unconditional weight update fraction: \{0.5, 0.7\}; the number of domains required for an update: \{3, 4\}. We used a beam size of 200, and limited the number of rule applications per derivation to 15.

For 7 domains, this setup requires training a parser 42 times ($7 \times 6$) for each hyper-parameter combination. The same process was used for tuning the hyper-parameters of each ablated model in the ablation analysis (see section 5.2), but the evaluation of the final parsers was on the training examples of the target domain (rather than its test examples).

**The Reported Metric** Following the standard practice, the metric we use is accuracy: the fraction of the test examples for which a correct denotation (desired state) was predicted. For examples where multiple logical forms achieve maximum score, we consider the fraction that have the correct denotation.

### 5.2 Results

Our main results are summarized in table 5.1. We compare three training methods: the standard AdaGrad and the two method described in chapter 4: CWU and TWOSTEP. The best average accuracy was achieved by TWOSTEP: 44.5%, a 5.4% absolute improvement over AdaGrad. CWU yielded an average accuracy of 39.6%, which is an absolute improvement of only 0.5% over AdaGrad.

In four domains TWOSTEP outperformed AdaGrad by more than 4%. The gap was most notable in the LIST and LIGHTING domains, where TWOSTEP outperformed by 14.3% and 12.4%, respectively. The improvements on MESSENGER and WORKFORCE were also substantial (8.4% and 4.1%, respectively). In the other three domains, TWOSTEP and AdaGrad performed identically (CONTAINER) or demonstrated differences of up to 2% (CALENDAR and FILE).

**Comparison with the In-Domain Setup** We compare the three training methods in zero-shot settings, trained with 603-611 training examples (i.e. all the training examples of the source domains) with AdaGrad in in-domain settings, trained with 96-104 examples from the target domain (i.e. all the training examples of the target domain). For the in-domain setup, hyper-parameters were tuned for each domain using 5-fold cross-validation on the training data, using the same grid search described in section 5.1. As shown in Table 5.1, the accuracy of AdaGrad with in-domain training is 15.2% higher than that of TWOSTEP with zero-shot training (59.7% vs. 44.5%), despite the smaller number of training examples. We did not experiment with the CWU and TWOSTEP methods in in-domain settings as they require multiple source domains.

**Model Ablation Analysis** We did a model ablation analysis, summarized in table 5.2, in which for each training method we compared the performance of our full model with two ablated model versions. The first ablated model does not use any signal form the application logic. That is, candidate logical forms that correspond to a method call that either does not
Table 5.1: Accuracy on the test set. In parentheses: results for the in-domain setup.

<table>
<thead>
<tr>
<th>Training Method</th>
<th>Model</th>
<th>CALENDAR</th>
<th>CONTAINER</th>
<th>FILE</th>
<th>LIGHTING</th>
<th>LIST</th>
<th>MESSENGER</th>
<th>WORKFORCE</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAGrad</td>
<td>full</td>
<td>38.0</td>
<td>38.1</td>
<td>28.0</td>
<td>36.2</td>
<td>44.5</td>
<td>53.3</td>
<td>38.8</td>
<td>39.1</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>39.0</td>
<td>38.1</td>
<td>(39.9)</td>
<td>39.4</td>
<td>88.0</td>
<td>(73.3)</td>
<td>57.7</td>
<td>(59.7)</td>
</tr>
<tr>
<td>CWU</td>
<td>full</td>
<td>35.9</td>
<td>32.0</td>
<td>28.7</td>
<td>36.2</td>
<td>46.8</td>
<td>58.9</td>
<td>38.8</td>
<td>39.6</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>38.0</td>
<td>38.1</td>
<td>28.7</td>
<td>48.6</td>
<td>58.8</td>
<td>61.7</td>
<td>42.9</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>-description</td>
<td>38.0</td>
<td>32.0</td>
<td>28.7</td>
<td>36.2</td>
<td>46.8</td>
<td>58.9</td>
<td>38.8</td>
<td>39.6</td>
</tr>
<tr>
<td>TWOStep</td>
<td>full</td>
<td>32.8</td>
<td>38.1</td>
<td>28.7</td>
<td>48.6</td>
<td>58.8</td>
<td>61.7</td>
<td>42.9</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>38.0</td>
<td>38.1</td>
<td>28.7</td>
<td>48.6</td>
<td>58.8</td>
<td>61.7</td>
<td>42.9</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>-description</td>
<td>38.0</td>
<td>32.0</td>
<td>28.7</td>
<td>36.2</td>
<td>46.8</td>
<td>58.9</td>
<td>38.8</td>
<td>39.6</td>
</tr>
</tbody>
</table>

Table 5.2: Ablation analysis on the training set using leave-one-out cross-validation over the domains.

<table>
<thead>
<tr>
<th>Training Method</th>
<th>Model</th>
<th>CALENDAR</th>
<th>CONTAINER</th>
<th>FILE</th>
<th>LIGHTING</th>
<th>LIST</th>
<th>MESSENGER</th>
<th>WORKFORCE</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAGrad</td>
<td>full</td>
<td>38.0</td>
<td>40.7</td>
<td>31.8</td>
<td>37.0</td>
<td>37.3</td>
<td>51.5</td>
<td>40.6</td>
<td>41.0</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>39.0</td>
<td>22.4</td>
<td>31.8</td>
<td>25.5</td>
<td>46.7</td>
<td>50.0</td>
<td>41.1</td>
<td>36.7</td>
</tr>
<tr>
<td></td>
<td>-description</td>
<td>38.0</td>
<td>37.8</td>
<td>36.3</td>
<td>6.3</td>
<td>40.4</td>
<td>55.7</td>
<td>28.7</td>
<td>33.5</td>
</tr>
<tr>
<td>CWU</td>
<td>full</td>
<td>38.0</td>
<td>34.9</td>
<td>32.5</td>
<td>37.0</td>
<td>41.5</td>
<td>62.5</td>
<td>38.6</td>
<td>40.7</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>38.0</td>
<td>18.6</td>
<td>30.5</td>
<td>24.5</td>
<td>47.2</td>
<td>58.3</td>
<td>41.1</td>
<td>37.3</td>
</tr>
<tr>
<td></td>
<td>-description</td>
<td>38.0</td>
<td>36.9</td>
<td>27.8</td>
<td>9.1</td>
<td>40.5</td>
<td>59.8</td>
<td>28.7</td>
<td>34.3</td>
</tr>
<tr>
<td>TWOStep</td>
<td>full</td>
<td>37.5</td>
<td>39.7</td>
<td>32.0</td>
<td>44.7</td>
<td>37.5</td>
<td>65.6</td>
<td>47.0</td>
<td>46.3</td>
</tr>
<tr>
<td></td>
<td>-application logic</td>
<td>40.0</td>
<td>22.4</td>
<td>30.0</td>
<td>24.5</td>
<td>38.4</td>
<td>63.5</td>
<td>43.1</td>
<td>37.4</td>
</tr>
<tr>
<td></td>
<td>-description</td>
<td>36.0</td>
<td>38.8</td>
<td>27.0</td>
<td>5.8</td>
<td>30.7</td>
<td>53.1</td>
<td>34.2</td>
<td>32.2</td>
</tr>
</tbody>
</table>

Error Analysis: We sampled 200 examples from the training set, sampling 10 examples per interface method. The following error analysis is for the TWOStep method, evaluated on this sample (in zero-shot settings, as described in section 5.1).

We found that 30.4% of the error is in incorrect utterances provided by the workers. In the other examples, we observed that the inference tended to yield top scoring candidate logical forms with incorrect relations and operations due to our model lacking features that could increase the score of the correct logical form. For example, consider the utterance “remove the longest container”, its correct logical form is:

\[ \text{removeContainers}(\text{argmax}(R[type], \text{ShippingContainer}, R[length])) \]

The word “longest” did not appear in any example in the source domains, and thus none of the relevant lexicalized features associated with argmax were useful. We found that 12.8%
of the error is due to incorrect parsing of utterances that reference an entity by its index, for example mapping the utterance “unload the container in terminal four” to the logical form unloadContainers(R[length].4) instead of unloadContainers(R[index].4).
Chapter 6

Conclusions and Future Directions

This work was motivated by the long term goal of allowing developers to enable a natural language user interface for their software with minimal overhead. We presented a novel task of parsing instructions from domains that are unseen during training into compositional logical forms, and introduced a new dataset and a framework for constructing and experimenting with such datasets. The task requires learning from weak supervision: the desired state of the application. We presented a solution to the task by extending the parser presented in [PL15] which was originally designed for question answering. We extracted additional features based on description phrases that are provided for the interface methods, and used the provided application logic to dismiss some of the incorrect candidate logical forms. We experimented with two training algorithms that are tailored for the zero-shot setup, one of which outperformed the AdaGrad baseline and achieved an average accuracy of 44.5% on the test set.

We hope this work will inspire readers to use our framework for collecting a larger dataset and experimenting with more approaches. Our framework is designed to allow integration with existing Java applications with minimal effort, making it easy to define new domains and collect annotated data.

**Future Directions** Possible future work includes using our framework to create a dataset with more domains, and experimenting with neural models (and perhaps synthesizing a large number of additional examples per domain). Additional interesting directions include extending our framework to support applications with more complicated data structures, and using documentation written by programmers. Also, it will be interesting to experiment with multi-utterance instructions, similarly to [LPL16] but in zero-shot settings.
Appendix A

Integration with Our Framework

Figure A.1 demonstrates the straight-forward process of integrating existing Java code with our framework. This allows to easily define new domains that can be experimented with, and collect annotated data for them. The required code for integration (i.e. the code that one would need to add to an existing Java code) is marked with an underline.
```java
public class LightingControlSystem implements NLIRootEntity {
    public List&lt;Room&gt; rooms = new LinkedList&lt;&gt;();

    @EnableNLI
    @NLIAnnotations(descriptions = {"turn on"})
    public void turnLightOn(Collection&lt;Room&gt; rooms) {
        rooms.forEach(r-&gt;r.lightMode = LightMode.ON);
    }

    @EnableNLI
    @NLIAnnotations(descriptions = {"turn off"})
    public void turnLightOff(Collection&lt;Room&gt; rooms) {
        rooms.forEach(r-&gt;r.lightMode = LightMode.OFF);
    }
}

public enum LightMode{
    ON, OFF
}

public class Room implements NLIEntity {
    public String roomName;
    public LightMode lightMode;
    public int floor;

    public LightMode getLightMode() {
        return lightMode;
    }

    public Room(String roomName, LightMode lightMode, int floor) {
        this.roomName = roomName;
        this.lightMode = lightMode;
        this.floor = floor;
    }
}
```

Figure A.1: The Java code for the LIGHTING domain, with the code required for integration marked with an underline.
Bibliography


The second training method of N, based on the separation of the training process into two stages: in the first stage, the weights are trained, as is done in the original method for the wide range of semantical fields, but using AdaGrad examples. In the second stage, AdaGrad uses a subset of the domains from the source. This method brought AdaGrad to improve the results relative to the previous methods. Furthermore, since the new method encourages researchers to use a new data set, it is more powerful and faster, and enables Java and other models, and a variety of APIs for the definition of new fields and easily obtain labelled examples for each model.

The second training method of N, based on the separation of the training process into two stages: in the first stage, the weights are trained, as is done in the original method for the wide range of semantical fields, but using AdaGrad examples. In the second stage, AdaGrad uses a subset of the domains from the source. This method brought AdaGrad to improve the results relative to the previous methods. Furthermore, since the new method encourages researchers to use a new data set, it is more powerful and faster, and enables Java and other models, and a variety of APIs for the definition of new fields and easily obtain labelled examples for each model.

The second training method of N, based on the separation of the training process into two stages: in the first stage, the weights are trained, as is done in the original method for the wide range of semantical fields, but using AdaGrad examples. In the second stage, AdaGrad uses a subset of the domains from the source. This method brought AdaGrad to improve the results relative to the previous methods. Furthermore, since the new method encourages researchers to use a new data set, it is more powerful and faster, and enables Java and other models, and a variety of APIs for the definition of new fields and easily obtain labelled examples for each model.
 Berkshire

Analyze semantic and amorphous text in natural language to formal language for interpretation. In many tasks, it is necessary to use semantic analysis to map questions in natural language to formal representations, and use the formal representation to answer the question. In other tasks, a software interface is given, and the natural language represents the instructions, and the software interface is responsible for organizing the requests and responses. In many past works, semantic analysis is used to create examples that are formalized in the representation, and not formal representations themselves, i.e., only to answer the question, or to create the results of executing an instruction. For testing, many works used examples from the field, i.e., the semantic analyzer is tested on examples from the field.

In this work, we present a new task: given a formal instruction in natural language, the task is to map the instruction to formal representation in the field. For this task, we assume that a group of fields is given, such as (example) an example of an application (for example, a calendar), and for each one there is a formal definition that includes the data of the application, the functions of the interface, and words that are expected to be mentioned in the formal representation.

The goal of this work is to train the semantic analyzer to identify examples from the field, and provide descriptions from the field (e.g., instead of the calendar function, enter the calendar function's name). For this goal, we divide the solution into two parts: the first is creating new training sets, as described in the Jakarta Shampoo tool. The second part is testing the model on the new training sets, and verifying the results.

We present two objectives for this work: the first is to create new training sets that are closer to reality. The second is to test new training sets on new data, i.e., to verify the results of the training. For this purpose, we present two approaches: stochastic gradient and AdaGrad. The first approach is to test new training sets on new data, i.e., to verify the results of the training. The second approach is to test new training sets on new data, i.e., to verify the results of the training. The second approach is to test new training sets on new data, i.e., to verify the results of the training.
המחקר נבע מהכיתונים של פרופסור מוריס אלקראל על רעיית מḤופשת לנדסמט תעשיות ו唛ות בפקולטה למדעי המחשב.

תודה

ברצוני להlarından על שילב, רועי אלקראל, אשר נתן לי חתימה אדית ללבט בשעון כוונת מחקר של קרן מלגות על שם מורה יונה עזראל, ולקרן מלגות של עיריית חיפה. על תמיכה כספית ונדיבות בהשתלמותי.

אני מודע לכלך מח売り על שמו מורי ואלהי עיראת והקרן מלגות של עיריית חיפה لتנשאות ובויהן על התמימה הקספית והדריכה בהשתלמותי.
ניתוח סמטישלהוראות
ללא דוגמאות אימונים בשחומתמטרה

ഹיבר על מחקר

לשם מולי חלקי של הדרישת לקבלת התואר
מוגיסר למדעים במדעי המחשב

עופר בבעלי

רוחש לגלנס הסקנונית --- מוקף טכנולוגייתワープ
טבת התשע”ע - חיפה - דצמבר 2018
ניתוח סנטשים של הוראות
ללא דוגמאות אימונים בתחום המטרה

עופר גבעולי